Multiple-Query Optimization for Materialized View Maintenance

by

José A. Blakeley
Computer Science Department
Indiana University

and

Héctor Hernández
Department of Computer Science
Texas A&M University

TECHNICAL REPORT NO. 267

Multiple-Query Optimization
for Materialized View Maintenance
by
José A. Blakeley and Héctor Hernández

January, 1989

Multiple-Query Optimization for Materialized View Maintenance

José A. Blakeley
Computer Science Department,
Indiana University

Héctor Hernández
Department of Computer Science,
Texas A&M University

January 22, 1989

Abstract

Materialized views are a useful mechanism to speed up query processing in a DBMS. In this paper, we formulate the problem of computing the set of changes required to incrementally update a materialized view as a special case of multiple-query optimization (mqo). We report our initial results on applying three mqo approaches to this problem: a simple approach, a state-space search approach, and a hybrid approach. In the simple approach, we prove that the structure of the queries in our problem let us reduce significantly the optimization cost. In the state-space search approach, we propose a judicious way of generating plans exploiting the special structure of our problem. In the hybrid approach, we show how a single-query optimizer can be integrated with one based on multiple-query decomposition. This is a step towards integrating seemingly disparate optimizers. As part of this approach, we also present an improved heuristic for multiple-query decomposition. Using a running example we have found that, for our problem, the simple approach may be good enough compared to the more complex state-space search approach, and that the hybrid approach may give significant savings as compared to the other two approaches.

Keywords: Query processing, Multiple-query optimization, Materialized views, Data caching

1 Introduction

Materialized views are *stored relations* resulting from the evaluation of a query on a set of base relations or views. They are a form of data caching that can be used to enhance query processing performance in a database system.

Materialized views can be used to enhance the performance of queries in distributed DBMSs. Today's computer communications networks give us access to many information retrieval systems. A common topology of these systems consists of a large network of workstations (or personal computers) connected to a mainframe that manages the access to a corporate database or information retrieval system (e.g., ADMS± [18]). These workstations have substantial computing power as well as large amounts of secondary storage available. Storing commonly accessed database views at the workstations may enhance the retrieval performance of these systems mainly because of reduced communication costs. It also relieves the mainframe from having to recompute a request repeatedly.

Materialized views can also be used to enhance the performance of some features available in extendible database management systems. They can be used as one of the alternatives to implement procedures as data types as proposed in the POSTGRES database system [22]. Materialized views can also be used as a mechanism to allow the timely execution of rules and queries in database systems supporting time-constrained applications [5].

However, performance enhancements provided by materialized views come at a cost. As the base relations in the database are updated, the materialized views may become out of date. Therefore, an efficient view maintenance mechanism must be provided. There are several methods to keep the materialized views up to date. The simplest, and perhaps the most expensive, is to recompute the expression defining the view every time the view is accessed after an update of the base relations, however, this is often unnecessary. A better method is to maintain the view incrementally [3,8,13,19]. The following example illustrates the idea of incremental maintenance of views.

Example 1.1 Consider the materialized view $v = \pi_X \sigma_C(r_1 \times r_2 \times r_3)$ and suppose that a transaction inserts sets of tuples ι_1 and ι_3 into relations r_1 and r_3, respectively. Let $r_1' = r_1 \cup \iota_1$ and $r_3' = r_3 \cup \iota_3$ represent the new state of relations r_1 and r_3 after the transaction is performed. The new state of the view, v', is $v' = \pi_X \sigma_C(r_1' \times r_2 \times r_3') = \pi_X \sigma_C(r_1 \times r_2 \times r_3) \cup \pi_X \sigma_C(r_1 \times r_2 \times \iota_3) \cup \pi_X \sigma_C(\iota_1 \times r_2 \times r_3) \cup \pi_X \sigma_C(\iota_1 \times r_2 \times \iota_3)$. Since $v = \pi_X \sigma_C(r_1 \times r_2 \times r_3)$ is the current state of the view we have $v' = v \cup \Delta v$ where $\Delta v = \pi_X \sigma_C(r_1 \times r_2 \times \iota_3) \cup \pi_X \sigma_C(\iota_1 \times r_2 \times r_3) \cup \pi_X \sigma_C(\iota_1 \times r_2 \times \iota_3)$. Thus, to collect the changes required to bring v up to date we need to "union" the results of three queries. Deletions and modifications can be handled in a similar way [3]. □

Some forms of materialization of data have been proposed before, for example, snapshots [1], quasi-copies [2], and evolving views [23]. However, none of these proposals suggest an efficient way of maintaining the materialized data current with the rest of the database. Materialized views have been used as a performance enhancement in the implementation of the Syntel programming

system which integrates capabilities of functional programming languages, database, expert, and spreadsheet systems [16]. Syntel does provide an incremental view maintenance mechanism.

A system may guarantee several degrees of currency in the kinds of materialized views it supports. Views may be kept *always current* with the rest of the database, that is, a view maintenance mechanism may be invoked as part of the commit of a transaction that updates the database. Views may be maintained *on demand*, that is, before the view is retrieved the maintenance mechanism is invoked. Views may also be maintained *periodically* (e.g., every 24 hours). The above degrees of currency determine the ends of a spectrum of possibilities. Quasi-copies are somewhere in the middle of the spectrum, they are a form of materialized data that is allowed to deviate from the database according to some criteria (e.g., no more than 10% error). In general, designing a DBMS that supports materialized data involves determining *when* to update them as well as *how* to update them. These two problems are technically different. However, once the "when" has been determined, the "how" can be performed by a unified mechanism used to maintain materialized views of diverse degrees of currency. Our paper addresses the "how" part of the problem.

Maintaining a materialized view incrementally typically requires three main steps: (1) keeping track of updates to the base relations, (2) computing the changes that need to be applied to the view, and (3) updating the materialized view. Gathering the changes needed to update the view may involve the computation of several queries. The number of queries to be computed depends on the number of base relations referenced in the view definition that have been updated since the latest view materialization. In this paper, we concentrate on stage (2) of the incremental approach to maintaining views defined by relational expressions using select, project, and join operators. We report the initial findings of our investigation on the use of multiple-query optimization (*mqo*) [4,7,15,17,21] to efficiently compute the changes to materialized views. In particular, we have studied three approaches to mqo. First, we propose a *simple approach* that uses the plans generated by a single-query optimizer on each of the queries in the set to be optimized and then shares common subexpressions among the different plans. Second, we propose a way of exploiting the particular structure of our special case of mqo to generate alternative plans for each of the queries in the set and then use a *state-space search* algorithm. Finally, we propose a *hybrid approach* that starts by applying multiple-query decomposition [4] to a multi-query graph representing the queries to be optimized, and ends by applying single-query optimization when the multi-query graph has been decomposed to the point where it consists only of connected components representing individual queries without common subexpressions. Using a running example, we compare their performance against a strategy that optimizes each query in the set individually. We have found that for our problem, the simple approach may be good enough compared to the more complex state-space

search approach. Also, there are indications that our hybrid approach may give significant savings when applied to our special mqo problem as compared to the other two approaches. In addition, the hybrid approach can be applied to the optimization of an arbitrary set of queries. Additional experimental research is required to substantiate statistically our findings.

The rest of the paper is organized as follows. Section 2 presents the notation and background as well as the formalization of the problem of computing the changes to the view as a mqo problem. Section 3 presents our simple approach. Section 4 presents a state-space search approach. Section 5 presents a hybrid approach, and Section 6 presents our conclusions.

2 Notation and Problem Statement

We shall follow standard notation [14,24] and only give some non-standard definitions here. We consider a database D consisting of a set of (base) relations $D = \{r_1, \ldots, r_n\}$ where each relation r_i is defined on a set of attributes R_i. In this paper, we deal exclusively with PSJ-expressions. A *PSJ-expression* is an expression of the form $Q = \pi_X \sigma_C(R_1 \times \cdots \times R_m)$, where C is a conjunction of atomic terms of the form $(A\ \theta\ B\ +\ c)$, $(A\ \theta\ c)$ where $\theta \in \{=, \neq, <, \leq, >, \geq\}$, A and B are elements of $R_1 \cup \cdots \cup R_m$, and c is a constant. We assume for the rest of this paper that atomic terms are of this form. Also, we assume that all the R_i's in the above Cartesian product are distinct (i.e., we do not allow self-joins), and all the attributes in the R_i's are also distinct. A *materialized view* is the (stored) relation v that results from the evaluation of a PSJ-expression $V = \pi_X \sigma_C(R_1 \times \cdots \times R_m)$ against the database D; i.e., $v = \pi_X \sigma_C(r_1 \times \cdots \times r_m)$.

2.1 The problem

In general, suppose we want to maintain the materialized view $v = \pi_X \sigma_C(r_1 \times r_2 \times \cdots \times r_p)$, and that a transaction updates $q \leq p$ relations. The updated view v' is given by $v' = v \sqcup \Delta v$, where $\sqcup$ denotes the simultaneous union and difference of tuples, and Δv is given by

$$
\begin{aligned}
\Delta v \;=\; & \pi_X \sigma_C(r_1 \times \cdots \times r_{p-q} \times r'_{p-q+1} \times \cdots \times r'_{p-2} \times r'_{p-1} \times \hat{r}_p) \;\cup \\
& \pi_X \sigma_C(r_1 \times \cdots \times r_{p-q} \times r'_{p-q+1} \times \cdots \times r'_{p-2} \times \hat{r}_{p-1} \times r'_p) \;\cup \\
& \pi_X \sigma_C(r_1 \times \cdots \times r_{p-q} \times r'_{p-q+1} \times \cdots \times r'_{p-2} \times \hat{r}_{p-1} \times \hat{r}_p) \;\cup \\
& \pi_X \sigma_C(r_1 \times \cdots \times r_{p-q} \times r'_{p-q+1} \times \cdots \times \hat{r}_{p-2} \times r'_{p-1} \times r'_p) \;\cup \\
& \cdots \;\cup \\
& \pi_X \sigma_C(r_1 \times \cdots \times r_{p-q} \times \hat{r}_{p-q+1} \times \cdots \times \hat{r}_{p-2} \times \hat{r}_{p-1} \times \hat{r}_p).
\end{aligned}
$$

Relations $r_j' = r_j - \delta_j$, $p - q + 1 \leq j \leq p$, where $\delta_j \subseteq \hat{r}_j$ is a set of tuples to be deleted from r_j. Relations $\hat{r}_i$ represent the *net* changes made to the corresponding relations within the transaction, i.e., $\hat{r}_i = \iota_i \cup \delta_i, \iota_i \cap \delta_i = \emptyset$, where ι_i is a set of tuples to be inserted into r_i. Relations $\hat{r}_i$ are assumed to be non-empty. If q relations are updated by a transaction, then Δv will consist of the union of $2^q - 1$ *PSJ*-expressions. The problem is how to compute the set of queries comprising Δv efficiently. In other words, how can we optimize the execution of the queries required to compute Δv collectively. This is a special case of a mqo problem because all the queries are defined by the same expression and the only differences are the operands on which the queries are performed.

Multiple query optimization is a natural approach to solving this problem for several reasons: (a) the set of queries to be optimized is known in advance, (b) since there is extensive relation overlap among the queries, many subexpressions can potentially be shared, and (c) since the set of queries share the same projection, selection, and join operations, all the potential common subexpressions might be equivalent and their detection becomes a simple pattern matching problem (i.e., we might not need to test subsumption of some expressions [6,10,12,17,21]; in Section 3 we prove that we do not need to test subsumption among selection tasks).

2.2 Approaches

There are two general alternatives to computing the set of queries in Δv. (a) To optimize and execute each of the expressions individually. This alternative is included as a basis for comparing the improvements provided by the mqo techniques discussed. (b) To optimize and execute the expressions in Δv as a set. Here we have several alternatives: (i) use a simple method that combines single query optimization with a careful utilization of common subexpressions, (ii) use a state-space search approach, and (iii) use a hybrid approach.

The following example is used to illustrate the various mqo algorithms presented throughout the paper.

Example 2.1 Consider four relation schemes $R_1(H, I)$, $R_2(J, K)$, $R_3(L, M)$, and $R_4(N, O)$ and $\Delta v = v_1 \cup v_2 \cup v_3$ where $v_1 = \sigma_{(H<10)\wedge(I=J)\wedge(K=L)\wedge(M=N)}(r_1 \times r_2 \times \hat{r}_3 \times r_4)$, $v_2 = \sigma_{(I=J)\wedge(K=L)\wedge(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$, and $v_3 = \sigma_{(I=J)\wedge(K=L)\wedge(M=N)}(\hat{r}_1 \times r_2 \times \hat{r}_3 \times r_4)$.

The expressions v_1, v_2, and v_3 are generated by the differential view maintenance approach for the case when only relations r_1 and r_3 have been updated since the latest materialization of the view. Relations $\hat{r}_1$ and $\hat{r}_3$ represent the sets of net changes on relations r_1 and r_3, respectively. For expressions v_2 and v_3 we do not show the term $(H < 10)$ of the view definition. We assume that this selection is automatically performed when the relations $\hat{r}_1$ and $\hat{r}_3$ are collected, before the

incremental view maintenance mechanism is invoked.

Let r be a relation; $|r|$ denotes the cardinality of the relation in number of pages. Let $r_1 \bowtie r_2$ be a join expression, sel_{12} (a number between 0 and 1) denotes the selectivity of the join as a fraction of tuples from $r_1 \times r_2$ that qualify for the join. Let $\sigma_C(r_1)$ be a select expression, scl_{11} denotes the selectivity of the select as a fraction of the pages from r_1 containing tuples that satisfy the select condition C. The cost of an expression is estimated as the sum of the number of pages read and written needed to compute the expression. We make the following assumptions: (1) all intermediate results of expressions are written to disk; (2) joins are computed using the nested loops method; (3) there are no indices on relations to be used when computing joins or selects; and (4) the cost of computing the union of results corresponding to the queries in Δv (e.g., v_1, v_2, and v_3 above) is ignored. The cost of performing a join is estimated as: $cost(r_1 \bowtie r_2)$ = no. of pages read + no. of pages written = $|r_1|*|r_2| + |r_1|*|r_2|*sel_{12}$. The cost of performing a select is estimated as: $cost(\sigma_C(r_1))$ = no. of pages read + no. of pages written = $|r_1| + |r_1| * sel_{11}$. Assume that $|r_1| = 1000$ pages, $|r_2| = 80$ pages, $|r_3| = 90$ pages, $|r_4| = 200$ pages, $|\hat{r}_1| = 1$ page, $|\hat{r}_3| = 2$ pages, $sel_{11} = 0.1$, $sel_{12} = 0.1$, $sel_{23} = 0.15$, and $sel_{34} = 0.03$.

The optimal local plans corresponding to v_1, v_2 and v_3 above with their costs are shown in Table 1. Thus, the total cost of computing the changes in Δv is given by $cost(v_1) + cost(v_2) + cost(v_3) = 42,231$. □

v_1	Cost(v_1)	v_2	Cost(v_2)	v_3	Cost(v_3)
$s_1 \leftarrow \sigma_{(H<10)}(r_1)$	1,100	$s_1 \leftarrow \sigma_{(I=J)}(\hat{r}_1 \times r_2)$	88	$s_1 \leftarrow \sigma_{(I=J)}(\hat{r}_1 \times r_2)$	88
$s_2 \leftarrow \sigma_{(M=N)}(\hat{r}_3 \times r_4)$	412	$s_2 \leftarrow \sigma_{(K=L)}(s_1 \times r_3)$	828	$s_2 \leftarrow \sigma_{(M=N)}(\hat{r}_3 \times r_4)$	412
$s_3 \leftarrow \sigma_{(K=L)}(r_2 \times s_2)$	1,104	$s_3 \leftarrow \sigma_{(M=N)}(s_2 \times r_4)$	22,248	$s_3 \leftarrow \sigma_{(K=L)}(s_1 \times s_2)$	111
$s_4 \leftarrow \sigma_{(I=J)}(s_1 \times s_3)$	15,840				
Total:	18,456	Total:	23,164	Total:	611

Table 1: Locally optimal plans and costs for v_1, v_2, and v_3.

3 A Simple Approach

In this section, we show that the common structure of the queries in Δv can be exploited to write simple mqo algorithms, which are more efficient than the general algorithms proposed so far. Simple mqo algorithms consist of two stages. In the first stage, single query optimization is applied to each of the input queries to obtain a set of individual optimal plans. In the second stage, common subexpressions are detected and shared to obtain a less expensive global plan; this stage

is expensive because involves testing for subsumption, e.g. see algorithm **IE** in [21]. Simple mqo algorithms are motivated by the fact that single query optimization is typically a good first-cut on the cost of executing a set of queries. This is illustrated by the following example.

Example 3.1 Consider the two queries $v_1 = r_1 \bowtie r_2 \bowtie \hat{r}_3 \bowtie \hat{r}_4$ and $v_2 = r_1 \bowtie r_2 \bowtie r_3 \bowtie \hat{r}_4$, where $|r_1| = |r_2| = |r_3| = 1000$ pages, and $|\hat{r}_3| = |\hat{r}_4| = 2$ pages. The join selectivities s_{12}, s_{23}, and s_{34} are all 0.001. Single query optimization yields the optimal nesting order $(r_1 \bowtie (r_2 \bowtie (\hat{r}_3 \bowtie \hat{r}_4)))$ for v_1 and $(r_1 \bowtie (r_2 \bowtie (r_3 \bowtie \hat{r}_4)))$ for v_2 with costs 2,007 and 6,006, respectively, for a total cost of 8,013. Using single query optimization it is not possible to share the subexpression $r_1 \bowtie r_2$ that appears in both v_1 and v_2. However, if we would insist on sharing it, we would obtain the nestings $((r_1 \bowtie r_2) \bowtie (\hat{r}_3 \bowtie \hat{r}_4))$ and $((r_1 \bowtie r_2) \bowtie (r_3 \bowtie \hat{r}_4))$ with a total cost of 1,006,010. Clearly, in this case single query optimization defeats multiple query optimization. □

In this section, we give a simple mqo algorithm for the queries in ΔV that does not need to check for subsumption of selection expressions, because we formally prove, under some very general assumptions about single-query optimizers, the following result: For any two queries Q_i and Q_j in ΔV, and for each pair of selection tasks in any local optimal plans for Q_1 and Q_2, the two tasks are either equivalent or non-comparable. We also prove the correctness of our simple mqo algorithm.

3.1 Tasks, plans, and costs

A *task* t is a statement of the form $t \equiv T = exp$, where T is a temporary relation or the keyword $RESULT$, that indicates that this task provides the result of the query, and exp is $\sigma_{C_1}(R_1)$, $\pi_X(R_1)$, or $R_1 \bowtie_{C_2} R_2$, where R_1 and R_2 are (base or temporary) relations. Given two tasks $t_1 \equiv T_1 = exp_1$ and $t_2 \equiv T_2 = exp_2$, such that T_1 and T_2 are union-compatible, we say that t_1 is *subsumed* by t_2, written $t_1 \Rightarrow t_2$, iff for any instances of the operand relations in exp_1 and exp_2, the result of evaluating exp_1 is contained in the result of evaluating exp_2; t_1 is *identical* (or *equivalent*) to t_2, written $t_1 \equiv t_2$, iff $t_1 \Rightarrow t_2$ and $t_2 \Rightarrow t_1$.

An *access plan for a query* Q is a sequence of tasks $< t_1 \equiv T_1 = exp_1, \ldots, t_{l-1} \equiv T_{l-1} = exp_{l-1}, t_l \equiv RESULT = exp_l >$ whose execution produces the answer to Q [21]. We represent the access plan $< t_1, \ldots, t_l >$ as an acyclic directed graph $P = (V, A, L)$, where V, A, and L, the sets of vertices, directed edges, and of vertex labels respectively, are defined as follows: for each task t_i there is a vertex v_i; (v_i, v_j) is a directed edge in A if the result of task t_i is used in task t_j; and for each $v_i \in V$, $L(v_i)$ is "$T_i = exp_i$" if task $t_i \equiv T_i = exp_i$. The cost to perform task t_i is $cost(t_i)$ = the number of pages read + the number of pages written. Also we define for all $v_i \in V$, $cost(v_i)$

$= cost(t_i)$, and the *cost of plan* $P(V, A, L)$ as $Cost(P(A, V, L)) = \Sigma_{v_i \in V}\ cost(v_i)$. We assume that we have a local optimizer PG that generates minimal-cost plans for any query as follows. Given a query Q, we shall denote by $PG(Q)$ *the plan for* Q *generated by* PG, where $PG(Q)$ is such that $Cost(PG(Q)) = min_{P_i \in \mathcal{P}_Q}\{Cost(P_i)\}$, and $\mathcal{P}_Q$ is the set of all possible plans that can be used to evaluate Q.

3.2 Assumptions about the plan generator

Let $Q = \pi_X \sigma_C(R_1 \times R_2 \times \cdots \times R_m)$ be a *PSJ*-expression. We assume that the graph $G = (V, A, L)$ for $PG(Q)$ is a tree. (This assumption is sound since we do not allow self-joins.) The root of the tree is the vertex whose label is $RESULT = exp$. The tree must be well-formed in the sense that for all $v \in V$, there is a directed path from v to the root of G. (Otherwise there would be redundant tasks in $PG(Q)$ that would violate the minimality of its cost.) Finally, we also make the assumption that for any selection task $t \equiv T = \sigma_{C^*}(R)$ (or join task $t \equiv T = R \bowtie_{C^*} S$) in $PG(Q)$ the condition C^* is a subcondition of the condition C of Q (i.e., the atomic terms in C^* are atomic terms in C).

3.3 σ-canonical plans

Let $Q = \pi_X \sigma_C(R_1 \times \ldots \times R_m)$ be a *PSJ*-expression, and let $R \subseteq R_1 \cup \ldots \cup R_m$. Let us assume that $C = C_1 \wedge \cdots \wedge C_l$, and let $C_{i_1}, \ldots, C_{i_k}$ be all the conjuncts in C that involve only attributes in R. Then C_R shall denote the conjunction $C_{i_1} \wedge \cdots \wedge C_{i_k}$. Intuitively, C_R is the strongest condition on R that can be defined using only conjuncts from C. We want to prove that under our cost-model there are some canonical plans for Q which we call σ-*canonical* plans. A plan P for Q is σ-*canonical* if for any selection task $t \equiv T = \sigma_{C'}(R)$ in P, $C' = C_R$. We prove that we can assume that PG gives us σ-canonical plans for any *PSJ*-expression by proving that given $PG(Q)$, we can convert it into a σ-canonical plan preserving minimality of cost and equivalence of plans. The next example shows that, in general, we may not have σ-canonical plans.

Example 3.2 Let us consider the query $Q = \pi_{CD}[\sigma_{(A>5)}(R_1(AB)) \bowtie_{(B=C)} R_2(CD)] \bowtie \sigma_{(A>10)}(R_1(AB))$, and assume $PG(Q)$ is the following plan: $T_1 = \sigma_{(A>5)}(R_1(AB))$, $T_2 = T_1 \bowtie_{(B=C)} R_2(CD)$, $T_3 = \sigma_{(A>10)}(T_1)$, $T_4 = \pi_{CD}(T_2)$, and $RESULT = T_3 \bowtie T_4$. In this query, $C_{R_1} = (A > 5) \wedge (A > 10)$, which is equivalent to $(A > 10)$. Then it is easy to see that if in this plan we replace in $T_1 = \sigma_{(A>5)}(R_1(AB))$ the condition $(A > 5)$ by the stronger condition $(A > 10)$, then we get a plan which is not equivalent to the original one. □

3.4 Equivalence of tasks for σ-canonical plans

The following lemma proves that, under the assumptions specified above, we can assume σ-canonicity for the plans generated by PG for any PSJ-expression.

Lemma 3.1 Let $Q = \pi_X \sigma_C(R_1 \times \cdots \times R_m)$ be a PSJ-expression, and let us consider $PG(Q)$. Let t^* be any selection task in $PG(Q)$ of the form $T^* = \sigma_{C^*}(R)$, where $R \subseteq R_1 \cup \cdots \cup R_m$. Let P' be $PG(Q)$ with t^* replaced by $t' \equiv T^* = \sigma_{C_R}(R)$, where C_R is as defined above. Then (a) $Cost(P') \leq Cost(PG(Q))$, (b) P' is an optimal plan, (c) $P' \equiv PG(Q)$.

Proof: (a) Let us consider $t^* \equiv T^* = \sigma_{C^*}(R)$ in $PG(Q)$ and $t' \equiv T^* = \sigma_{C_R}(R)$. Since our cost model only takes into account page accesses, $cost(t') \leq cost(t^*)$ (because t' needs no more pages than t^* to write T^*). Hence $Cost(P') \leq Cost(PG(Q))$.

(b) Notice that $Cost(P') = Cost(PG(Q))$ since $Cost(PG(Q)) \leq Cost(P')$ by definition of $PG(Q)$. Therefore, we still have an optimal plan if we replace t^* by t'. This means P' is just another optimal plan. We now prove P' is equivalent to $PG(Q)$.

(c) Let $G = (V, A, L)$ be the graph for the access plan $PG(Q)$. From our assumptions about PG, G is a tree. Let us assume that v_l is the vertex in V whose corresponding task is $t_l \equiv RESULT = exp$; i.e., v_l is the root of G. Let t^*, t', and P' be as defined above. Remember that from our assumptions about PG, the condition C^* in t^* *must* be a subcondition of the condition C_R in t'. Assume that C^* is a proper subcondition of C_R; otherwise our proof is complete.

Let v^* be the vertex in G that corresponds to t^*. From: (i) G is a tree, (ii) all attributes and base relations are distinct, and (iii) the query satisfies C_R, it is not difficult to prove that in the path from v^* to v_l or in the paths from the leaf vertices of G to v^* there must be selection tasks whose conditions include the conjuncts in $C_R - C^*$, which denotes the conjunction of atomic terms in C_R but not in C^*. Then replacing t^* by t' is equivalent to replicating upwards or downwards (relative to v^*) $\sigma_{C_R - C^*}$, which produces an equivalent expression for Q. Therefore $P' \equiv PG(Q)$. □

In view of the above result, in the rest of this section we assume that the plan generated by PG for any PSJ-expression is a σ-canonical plan. That is, for any PSJ-expression Q, $PG(Q)$ is such that $Cost(PG(Q)) = min_{P_i \in \mathcal{P}_Q}\{Cost(P_i)\}$, where $\mathcal{P}_Q$ is the set of all possible σ-canonical plans that can be used to evaluate Q. The following lemma states that we do not need to detect subsumption among selection tasks to obtain a global optimal plan to compute ΔV, if we use a simple mqo algorithm.

Lemma 3.2 Let $P_1 = PG(Q_1)$ and $P_2 = PG(Q_2)$, where Q_1 and Q_2 are any two queries in ΔV. Let t_1 be a selection task of the form $T_1 = \sigma_{C_1}(S_1)$ in P_1 and let t_2 be a selection task of the form $T_2 = \sigma_{C_2}(S_2)$ in P_2, where the tasks that generate S_1 and S_2 are equivalent. Then $t_1 \equiv t_2$.

Proof: Since the tasks that generate S_1 and S_2 are equivalent, it must be the case that they are union-compatible. Then let $S = S_1 = S_2$. Then since both Q_1 and Q_2 are defined using exactly the same condition C, and since we are assuming PG is generating σ-canonical plans for any PSJ-expression, $C_{S_1} = C_{S_2} = C_S$. Therefore, since the tasks that generate S_1 and S_2 are equivalent, t_1 must be equivalent to t_2. □

3.5 A simple mqo algorithm

We present below an algorithm where according to Lemma 3.2 we do not have to check subsumption between selection tasks. This algorithm can also be used to merge equivalent tasks of a set of plans whose graphs are trees that do not contain redundant tasks.

Algorithm S. A simple mqo algorithm to obtain an optimal plan to compute ΔV.
Input: A set of queries $\Delta V = \{V_1, \ldots, V_{2^q-1}\}$, where each V_j is a PSJ-expression of the form $\pi_X(\sigma_C(R_{j_1} \times R_{j_2} \times \cdots \times R_{j_p}))$.
Output: A directed, acyclic, labeled graph $GP = (GV, GA, GL)$ that represents a global optimal plan to compute ΔV.
Comments: See the above text for notation and assumptions.
Method:

1. *Obtain local optimal plans for each $Q_i \in \Delta v$.* For $1 \leq i \leq 2^q - 1$, let $G_i = (V_i, A_i, L_i)$ be the graph for $PG(Q_i)$; we assume the vertices in all the V_i's are distinct.

2. *Initialization of GP.* Let $GP = (GV, GA, GL)$, where GV, GA, and GL are defined as follows: Let $l = 2^q - 1$; then $GV = \bigcup_{i=1}^{l} V_i$; $GA = \bigcup_{i=1}^{l} A_i$; and for $1 \leq i \leq l$, and for each $v_j \in V_i$, $GL(v_j) = L_i(v_j)$.

3. *List the vertices in each V_i in some order.* For $1 \leq i \leq 2^q - 1$, let $\mathcal{L}_i$ be the list of the vertices of $G_i = (V_i, A_i, L_i)$ in *postorder* (e.g., $\mathcal{L}_i = postorder(root_i)$, where $root_i$ is the vertex in $G_i = (V_i, A_i, L_i)$ whose label is $RESULT_i = exp$); assume $\mathcal{L}_i =< v_i^1, \ldots, v_i^{j_i} >$, where $j_i = |V_i|$; then for any pair v_i^r, v_i^s of vertices in $\mathcal{L}_i$, (we say that) $v_i^r \prec v_i^s$ if $r < s$.

4. *Orderly Merge of Identical Tasks.* For $1 \leq i \leq (2^q - 1) - 1$, do the following.

4.1 For $1 \leq l \leq |\mathcal{L}_i|$ do:

4.1.1 *Obtain the tasks in plans $i+1, \ldots, 2^q - 1$ equivalent to t_i^l.* Let t_i^l be the task corresponding to v_i^l. For $i + 1 \leq j \leq 2^q - 1$, if there exists a vertex in $\mathcal{L}_j$ such that its corresponding task is equivalent to t_i^l, then let v_j^* be the smallest (according to $\prec$) vertex in $\mathcal{L}_j$ such that its corresponding task t_j^* is equivalent to t_i^l. Let $Eq\_tasks =< v_{j_1}^*, \ldots, v_{j_k}^* >$ be the

vertices whose corresponding tasks are equivalent to t_i^l, in the order in which they were obtained; if *Eq_tasks* is the empty sequence, then we go to 4.2. Let the corresponding tasks of $v_{j_1}^*, \ldots, v_{j_k}^*$ be $t_{j_1}^*, \ldots, t_{j_k}^*$.

4.1.2 *Mark redundant tasks and their outgoing edges.* For $1 \leq r \leq k$, mark in GP "for-deletion" both the vertex $v_{j_r}^*$ and each edge $(v_{j_r}^*, v')$ in GA, for any vertex v'; we assume, without loss of generality that $v_{j_r}^*$ is not the result vertex of the plan j_r.

4.1.3 *Mark the predecessors of redundant tasks and the edges connecting them.* For each vertex $v_{j_r}^*$, $1 \leq r \leq k$ do: Let V' be the set of all the vertices v in GP that come from V_{j_r}, the graph from where $v_{j_r}^*$ comes from, such that there is a directed path from v to $v_{j_r}^*$; if $V' = \emptyset$ then mark in GP "for-deletion" the incoming edges of v_{j_r}, else mark in GP "for-deletion" the vertices in V' along with all their incoming and outgoing edges.

4.1.4 *Fixing GP and $\mathcal{L}_i$'s.* For $1 \leq r \leq k$, for each edge $(v_{j_r}^*, v')$ in GA, where v' is some vertex in GV, add the edge (v_i^l, v') to GA, and replace the relation $T_{j_r}^*$ by T_i^l in $GL(v')$, where $T_{j_r}^*$ is the relation in $t_{j_r}^* \equiv T_{j_r}^* = exp_{j_r}^*$ and T_i^l is the relation in $t_i^l \equiv T_i^l = exp_i^l$. Remove from the graph GP all the vertices and edges marked "for-deletion;" also remove from the $\mathcal{L}_i$'s the vertices removed from GP.

4.2 **end of for-loop that begins at Step 4.1** /* $l := l + 1$ */

5. **end of for-loop that begins at Step 4** /* $i := i + 1$ */

3.6 Correctness of Algorithm S

In this subsection, we prove that Algorithm S is correct. First, since we are deleting edges from GP, we must prove that at the end of each iteration of the **for-loop** that begins at Step 4.1 and ends at Step 4.1.4, GP still represents a *complete plan*; i.e., each vertex in GP has an incoming edge for each input that its corresponding task needs for evaluating its expression. Secondly, it has to be proven that GP represents a plan which is equivalent to ΔV. Thirdly, we have to prove that the graph output by Algorithm S is acyclic. We prove the plan represented by GP is complete and we also prove the acyclicity of GP. Its equivalence to ΔV is straight forward.

Lemma 3.3 Let $\Delta v = \{v_1, \ldots, v_{2^q-1}\}$ be the input to Algorithm S, and consider an execution of Algorithm S with this input. For $1 \leq i \leq 2^q - 1$, let $G_i = (V_i, A_i, L_i)$ be the graph for $PG(Q_i)$. Let $GP = (GV, GA, GL)$, where $GV = \bigcup_{i=1}^{l} V_i$, $GA = \bigcup_{i=1}^{l} A_i$, and $l = 2^q - 1$. Let GP_{jk}, GV_{jk}, and GA_{jk} be the values of GP, GV, and GA before the k-th execution of the **for-loop** that begins at Step 4.1, when the value of the variable i at Step 4 is j (i.e., $i = j$); observe that $GP_{11} = GP$, $GV_{11} = GV$, and $GA_{11} = GA$. Then for $1 \leq j \leq 2^q - 2$, and for $1 \leq k \leq |\mathcal{L}_j|$, where $\mathcal{L}_j$ is the list of the vertices of V_j in postorder, if GP_{jk} represents a complete plan, then after the execution of Steps 4.1.1 through 4.1.4 (inclusive) GP_{jk} still represents a complete plan.

Proof: Assume GP_{jk} represents a complete plan. Then each vertex in GP_{jk} has an incoming edge for each input that its corresponding task needs for evaluating its expression.

Let us consider the edges marked "for-deletion" from GP_{jk} at Steps 4.1.2. and 4.1.3. First, it should be clear that the edges deleted at Step 4.1.2 are replaced by the ones inserted at Step 4.1.4.

For the edges marked for deletion in Step 4.1.3, let us assume that plan m, $m > j$, has a vertex v_m^* whose corresponding task is equivalent to t_j^k. Let v' be a vertex in GP_{jk} from V_m, the set of vertices from where v_m^* comes, whose edges are marked "for deletion." The deletion of any of its incoming edges does not affect the completeness of the graph, because the vertex itself is deleted as well. Let us now consider the deletion of any of its outgoing edges (v', v''), for some v'' in GV_{jk}. There are two cases to be considered depending on whether v'' belongs to V_m. If $v'' \in V_m$, then there is no problem, since v'' is also marked "for deletion" in Step 4.1.3 (because by assumption about PG, $G_m = (V_m, A_m, L_m)$ is a tree and it does not have redundant tasks imply that either v'' is v_m^* or there is a path from v'' to v_m^*, since (v', v'') must be in the unique path from v' to v_m^*). The case $v'' \notin V_m$ it is not possible, because up to this point in the execution of Algorithm S we have not added any new outgoing edge to any plan whose index is greater than j. This completes our proof. □

Now we prove that the graph output by Algorithm S is indeed acyclic.

Lemma 3.4 The output from Algorithm S is an acyclic graph.

Proof: The key observation is that if we add to GA a new (directed) edge that connects $v_i \in V_i$ with a vertex $v_j \in V_j$, then $i < j$. Then it should not be difficult to see that, since we start with acyclic G_i's (by assumption about PG), and we only add edges that connect vertices in distinct V_i's always in the direction of increasing indices of G_i's, there is no way a cycle can be formed. □

The following example illustrates how Algorithm S works.

Example 3.3 Let us apply the above algorithm to the three queries in Δv of Example 2.1: $v_1 = \sigma_{(H<10)(I=J)(K=L)(M=N)}(r_1 \times r_2 \times \hat{r}_3 \times r_4)$, $v_2 = \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$, and $v_3 = \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times \hat{r}_3 \times r_4)$. Step 1 of Algorithm S obtains optimal access plans for each of the three queries. The plans are $P_1 = PG(v_1) = < t_1^1 \equiv \sigma_{(H<10)}(r_1),\ t_1^2 \equiv \sigma_{(M=N)}(\hat{r}_3 \times r_4),\ t_1^3 = \sigma_{(K=L)}(r_2 \times t_1^2),\ t_1^4 \equiv \sigma_{(I=J)}(t_1^1 \times t_1^3) >$, $P_2 = PG(v_2) = < t_2^1 \equiv \sigma_{(I=J)}(\hat{r}_1 \times r_2),\ t_2^2 \equiv \sigma_{(K=L)}(t_2^1 \times r_3),\ t_2^3 \equiv \sigma_{(M=N)}(t_2^2 \times r_4) >$, $P_3 = PG(v_3) = < t_3^1 \equiv \sigma_{(I=J)}(\hat{r}_1 \times r_2),\ t_3^2 \equiv \sigma_{(M=N)}(\hat{r}_3 \times r_4),\ t_3^3 \equiv \sigma_{(K=L)}(t_3^1 \times t_3^2) >$. The equivalent tasks are $t_1^2 \equiv t_3^2$, $t_2^1 \equiv t_3^1$.

After processing the first plan P_1, task t_3^2 is removed from P_3 and it is replaced by t_1^2 in the expression for t_3^3. So P_3 becomes $< t_3^1 \equiv \sigma_{(I=J)}(\hat{r}_1 \times r_2),\ t_3^3 \equiv \sigma_{(K=L)}(t_3^1 \times t_1^2) >$. After processing

the tasks in the second plan task t_3^1 is removed from P_3 and it is replaced by t_2^1 in the expression for t_3^3. The final value of P_3 is $< t_3^3 \equiv \sigma_{(K=L)}(t_2^1 \times t_1^2) >$. Since, from the tables in Example 2.1, $Cost(P_1) = 18{,}456$, $Cost(P_2) = 23{,}164$, and $cost(t_3^3) = 111$, the cost of this global plan is 41,731 which provides savings of 1.2% over the solution with no sharing of common subexpressions (i.e., 42,231, see Example 2.1). We will see that, for this example, the solution given by the simple approach is identical to the solution given by the state-space search approach of next section. □

3.7 Remarks

Algorithm S works with the lists $\mathcal{L}_i$'s in any order (postorder, inorder, depth-first, arbitrary order, etc.). We chose postorder because we felt this order is best for our particular mqo problem. The fact that testing subsumption among selection tasks is not required provides a substantial improvement on the cost of optimization. In the worst case we have savings in the order of $\prod_{i=1}^{m} |V_i| \times k$, where $m = |\Delta V|$, $|V_i|$ is the number of tasks that comprises the i-th FSJ-expression in ΔV, and k is the time required to test for subsumption of selection tasks (which in general requires a theorem prover). As we mentioned above, Algorithm S can be applied to any set of queries as long as their plans satisfy our assumptions about the plans generated by PG. The choices for relation cardinalities and predicates selectivities for this example do not yield big savings, but this does not necessarily mean mqo is not worth it. We are using this example as a measure of relative comparison among the approaches. Finally, this section has shown that it is important to understand the properties of the plans generated by the optimizer in order to find ways of reducing optimization costs.

4 A State-Space Search Approach

Many single- and multiple-query optimization algorithms (e.g., [7,15,20]) include a stage that performs a search of the state-space defined by the alternative plans that could be used to execute the query or queries. In this sense, all query optimizers that at some point perform a search of the solution space can be classified within this approach. However, in this section we use the term *state-space search* more narrowly to mean mqo algorithms whose solution guarantees some optimality of the global plan generated. A multiple-query optimizer in this approach includes the following stages: (1) generate alternative plans per query, (2) identify equivalent tasks among the strategies generated in the previous stage, (3) estimate costs per task and plans, and (4) use a state-space search algorithm (e.g, exhaustive search, branch and bound [7], dynamic programming [15], A* [21]) to find a globally optimal plan.

Previous mqo research following the state-space search approach has concentrated on stages (3) and (4) of the approach and has assumed that the generation of plans and identification of equivalent tasks is performed by some other component of the optimizer. *Plan generation* is the focus in this section. We propose to exploit a characteristic of our mqo problem, namely, *all queries in ΔV are queries represented by very similar (if not the same) relational expression.* Two queries in ΔV differ only in their operand relations. Because of their difference in operands, locally optimal plans (at the logical level) generated for two queries in ΔV may be different. However, *each locally optimal plan is a feasible plan (i.e., a plan template) for any query in ΔV, provided the appropriate operands are used.* Therefore, we propose to use all distinct locally optimal plans generated for the $2^q - 1$ queries in ΔV as the set of alternative plans for each query used in stage (1) of the approach.

Before we continue our discussion we need to introduce some terminology commonly used by research in the state-space search approach. Using Grant and Minker's notation, we are given a set of queries v_i, $1 \leq i \leq n$, whose evaluation is to be optimized globally; the plans P_{ij} (a set of tasks), $1 \leq j \leq p_i$, for evaluating each query v_i; the distinct atomic tasks t_{ij}^k, $1 \leq k \leq q_{ij}$, which comprise each plan P_{ij}; and the actual or estimated cost $cost(t_{ij}^k)$ for each task. Equivalent tasks among plans are assumed to be known. The objective is to find a sequence of plans $\langle P_{1k_1}, P_{2k_2}, \ldots, P_{nk_n} \rangle$, whose cost is minimal.

Consider a solution vector $S_k = \langle P_{1k_1}, P_{2k_2}, \ldots, P_{nk_n} \rangle$. Since S_k is consider to be a global plan, its cost is given by $cost(S_k) = \sum_{t \in \bigcup_{i=1}^{n} P_{ik_i}} cost(t)$. The *coalesced cost* on tasks [7] is given by $coalesced\_cost(t) = \frac{cost(t)}{n_q}$, where n_q is the number of queries in which task t occurs. Similarly, for plans we have $coalesced\_cost(P_{ij}) = \sum_{k=1}^{q_{ij}} coalesced\_cost(t_{ij}^k)$. A more formal description of the state-space search approach to mqo is given below.

Algorithm SS
Input: a set of queries v_i, $1 \leq i \leq n$.
Output: a sequence of plans $\langle P_{1k_1}, P_{2k_2}, \ldots, P_{nk_n} \rangle$, whose cost is minimal.
Method:

1. For each query v_i, $1 \leq i \leq n$, find the optimal execution plan. Identify the tasks t_{ij}^k along with their costs $cost(t_{ij}^k)$, $1 \leq k \leq q_{ij}$, comprising each plan.

2. For each query v_i generate alternative plans P_{ij} using each different plan obtained is Step 1. At this step identify all equivalent tasks among the different plans, and find the "actual" cost associated with each task.

3. Compute the estimated costs for tasks and plans in the optimal solution vector (e.g., coalesced costs).

4. Use a search algorithm to find a solution vector of minimal cost. □

Example 4.1 Let us apply the above algorithm to the three queries in ΔV of Example 2.1. Step 1 of Algorithm SS finds the locally optimal access plans for each of the three queries; see Table 1. Apart from the selection on relation r_1 required in query v_1, the locally optimal access plans provide three different nesting orders for performing the joins. Thus, we have three alternative plans (templates) that we can use for each of the queries in ΔV. The plans to consider are given in Table 2. The equivalent tasks are $t^2_{11} \equiv t^3_{13} \equiv t^1_{31} \equiv t^2_{33}$, $t^3_{11} \equiv t^2_{31}$, and $t^1_{22} \equiv t^1_{23} \equiv t^1_{32} \equiv t^1_{33}$. Table 3 shows estimated costs for tasks and plans and Table 4 shows their corresponding coalesced costs.

Plans	Tasks			
P_{11}	$t^1_{11} \equiv \sigma_{(H<10)}(r_1)$	$t^2_{11} \equiv \sigma_{(M=N)}(r_3 \times r_4)$	$t^3_{11} \equiv \sigma_{(K=L)}(r_2 \times t^2_{11})$	$t^4_{11} \equiv \sigma_{(I=J)}(t^1_{11} \times t^3_{11})$
P_{12}	$t^1_{12} \equiv \sigma_{(H<10)}(r_1)$	$t^2_{12} \equiv \sigma_{(I=J)}(t^1_{12} \times r_2)$	$t^3_{12} \equiv \sigma_{(K=L)}(t^2_{12} \times r_3)$	$t^4_{12} \equiv \sigma_{(M=N}(t^3_{12} \times r_4)$
P_{13}	$t^1_{13} \equiv \sigma_{(H<10)}(r_1)$	$t^2_{13} \equiv \sigma_{(I=J)}(t^1_{13} \times r_2)$	$t^3_{13} \equiv \sigma_{(M=N)}(r_3 \times r_4)$	$t^4_{13} \equiv \sigma_{(K=L}(t^2_{13} \times t^3_{13})$
P_{21}	$t^1_{21} \equiv \sigma_{(M=N)}(r_3 \times r_4)$	$t^2_{21} \equiv \sigma_{(K=L)}(r_2 \times t^1_{21})$	$t^3_{21} \equiv \sigma_{(I=J)}(r_1 \times t^2_{21})$	
P_{22}	$t^1_{22} \equiv \sigma_{(I=J)}(r_1 \times r_2)$	$t^2_{22} \equiv \sigma_{(K=L)}(t^1_{22} \times r_3)$	$t^3_{22} \equiv \sigma_{(M=N)}(t^2_{22} \times r_4)$	
P_{23}	$t^1_{23} \equiv \sigma_{(I=J)}(r_1 \times r_2)$	$t^2_{23} \equiv \sigma_{(M=N)}(r_3 \times r_4)$	$t^3_{23} \equiv \sigma_{(K=L)}(t^1_{23} \times t^2_{23})$	
P_{31}	$t^1_{31} \equiv \sigma_{(M=N)}(r_3 \times r_4)$	$t^2_{31} \equiv \sigma_{(K=L)}(r_2 \times t^1_{31})$	$t^3_{31} \equiv \sigma_{(I=J)}(r_1 \times t^2_{31})$	
P_{32}	$t^1_{32} \equiv \sigma_{(I=J)}(r_1 \times r_2)$	$t^2_{32} \equiv \sigma_{(K=L)}(t^1_{32} \times r_3)$	$t^3_{32} \equiv \sigma_{(M=N)}(t^2_{32} \times r_4)$	
P_{33}	$t^1_{33} \equiv \sigma_{(I=J)}(r_1 \times r_2)$	$t^2_{33} \equiv \sigma_{(M=N)}(r_3 \times r_4)$	$t^3_{33} \equiv \sigma_{(K=L)}(t^1_{33} \times t^2_{33})$	

Table 2: Alternative plans for queries v_1, v_2, and v_3.

Plans	Tasks				Total
P_{11}	$t^1_{11} = 1,100$	$t^2_{11} = 412$	$t^3_{11} = 1,104$	$t^4_{11} = 15,840$	18,456
P_{12}	$t^1_{12} = 1,100$	$t^2_{12} = 8,800$	$t^3_{12} = 1,840$	$t^4_{12} = 49,440$	61,180
P_{13}	$t^1_{13} = 1,100$	$t^2_{13} = 8,800$	$t^3_{13} = 412$	$t^4_{13} = 11,040$	21,352
P_{21}	$t^1_{21} = 18,540$	$t^2_{21} = 49,680$	$t^3_{21} = 7,128$		75,348
P_{22}	$t^1_{22} = 88$	$t^2_{22} = 828$	$t^3_{22} = 22,248$		23,164
P_{23}	$t^1_{23} = 88$	$t^2_{23} = 18,540$	$t^3_{23} = 4,968$		23,596
P_{31}	$t^1_{31} = 412$	$t^2_{31} = 1,104$	$t^3_{31} = 159$		1,675
P_{32}	$t^1_{32} = 88$	$t^2_{32} = 19$	$t^3_{32} = 618$		725
P_{33}	$t^1_{33} = 88$	$t^2_{33} = 412$	$t^3_{33} = 111$		611

Table 3: Costs for tasks and plans.

Using Sellis' algorithm, the solution vector for the above problem is given by $\langle P_{11}, P_{22}, P_{33} \rangle$ with a cost of 41,179 which apparently provides savings of 2.4% over the solution with no sharing of common subexpressions (i.e., 42,231). Although the costs obtained by Algorithm S and Algorithm SS appear to be different they are not so. The global plans obtained by the two methods are the same. The difference in cost comes from the fact that Algorithm SS works with coalesced costs. In

Plans	Tasks				Total
P_{11}	$t^1_{11} = 1,100$	$t^2_{11} = 206$	$t^3_{11} = 552$	$t^4_{11} = 15,840$	17,698
P_{12}	$t^1_{12} = 1,100$	$t^2_{12} = 8,800$	$t^3_{12} = 1,840$	$t^4_{12} = 49,440$	61,180
P_{13}	$t^1_{13} = 1,100$	$t^2_{13} = 8,800$	$t^3_{13} = 206$	$t^4_{13} = 11,040$	21,146
P_{21}	$t^1_{21} = 18,540$	$t^2_{21} = 49,680$	$t^3_{21} = 7,128$		75,348
P_{22}	$t^1_{22} = 44$	$t^2_{22} = 828$	$t^3_{22} = 22,248$		23,120
P_{23}	$t^1_{23} = 44$	$t^2_{23} = 18,540$	$t^3_{23} = 4,968$		23,552
P_{31}	$t^1_{31} = 206$	$t^2_{31} = 552$	$t^3_{31} = 159$		917
P_{32}	$t^1_{32} = 44$	$t^2_{32} = 19$	$t^3_{32} = 618$		681
P_{33}	$t^1_{33} = 44$	$t^2_{33} = 206$	$t^3_{33} = 111$		361

Table 4: Coalesced costs for tasks and plans.

this case, we used the coalesced cost 552 for task t^2_{31}. But this task is not shared among the plans, therefore, its cost must be 1,104. This accounts for the cost difference between the results obtained by Algorithms S and SS. Hence, for this example, the savings obtained by the state-space search approach are the same as the ones obtained by the simple approach. □

The following question arises at this point: *Is it cost-effective to use a state-space search approach to compute* ΔV*?* In many cases, the greatest reduction in cost when optimizing multiple queries comes from the individual optimization of each of the queries. Thus, adding a stage that detects and utilizes common subexpressions after local optimization of individual queries in ΔV (i.e., Algorithm S) may be good enough. Spending a lot of effort generating arbitrary plans per query and performing the state-space search of an optimal plan adds a substantial portion to the optimization cost and may not be cost effective in the computation of ΔV. Initial experiments with Ingres comparing the relative costs of computing ΔV using the simple algorithm and a state-space search approach show that often both methods yield similar results.

5 A Hybrid Approach

This algorithm is partly inspired by the multiple-query decomposition algorithm of Chakravarthy and Minker [4] which is a generalization of query decomposition proposed by Wong and Youssefi [25]. The approach is a *hybrid* from multiple-query decomposition and single-query optimization. It is formed from two stages. In the first stage the set of queries is decomposed using a modified version of multiple-query decomposition until no more common subexpressions can be shared. That is, the multi-query graph [4] representing the set of queries has been decomposed so that it consists only of connected components representing individual queries. At that point, the second stage of

the algorithm is activated. The algorithm switches to a single-query optimization strategy that is better than query decomposition. Each of the connected components representing a subexpression for a single query is "fully optimized" (e.g., an optimal nesting order for joins is computed).

5.1 The hybrid algorithm

We assume the reader is familiar with the notion of *instantiation* and *iteration* used in query decomposition approaches [14,24,25] and with the multiple query decomposition algorithm of Chakravarthy and Minker [4].

A problem that we identify in query decomposition algorithms previously proposed [4,14] is that they are nondeterministic in the sense that the heuristics proposed are not precise about when instantiation or iteration should be applied. As a consequence, when these algorithms are invoked using as input two isomorphic graphs for a given query (or set of queries) the algorithms may yield distinct execution programs for the query.

Algorithm II described below removes this problem by proposing a more specific set of heuristics using cost estimations as functions of predicate selectivities and cardinalities of relations to determine a *unique* order in which iteration and instantiation are applied. Thus, for a fixed set of cost estimation functions, any two isomorphic graphs representing a given set of queries will be decomposed into a *unique* execution program.

Let $G = (V, E)$ denote an undirected multi-query graph. We abuse the notation to denote an edge $e \in E$ as a four-tuple (r, r', j, ρ) where r and r' are the two nodes connected by the edge, j is a number (also called the color) that identifies the query to which the edge belongs, and ρ represents the label (condition) of the edge. Let $adjacent(r)$ be the set of vertices adjacent to node r, that is, $adjacent(r) = \{r' \mid (\exists\, j, \rho)\ (r, r', j, \rho) \in E, 1 \leq j \leq 2^q - 1\}$. We remind the reader that $2^q - 1$ is the number of queries that form ΔV. Let $edges(r, r')$ be the set of all edges between nodes r and r', that is, $edges(r, r') = \{e \mid (\exists\, j, \rho)\ e = (r, r', j, \rho) \in E, 1 \leq j \leq 2^q - 1\}$. Let $\nu(r, r')$ be a function defined as follows:

$$\nu(r, r') = \begin{cases} 0 & \text{if } |edges(r, r')| = 0 \\ |edges(r, r')| - 1 & \text{if } |edges(r, r')| \geq 1 \end{cases}$$

We define a function $savings(r)$ to be the savings in cost provided by iterating on relation r as follows: $savings(r) = \sum_{r' \in adjacent(r)} cost(r \bowtie r') * \nu(r, r')$.

Briefly, what the function *savings* is trying to capture is the following. Suppose that there is a pair of nodes connected by three edges representing the same predicate, meaning that there are three queries sharing the same subexpression. If the system computes the expression defined by

one of these edges, then the system will not need to compute the same expression for the other two queries again (provided the result is stored for later use), which basically saves twice the cost of computing the expression. Savings are obtained when there are at least two edges representing the same predicate between two nodes.

Algorithm H. A hybrid algorithm between multiple-query decomposition and single-query optimization.
Input: a set of queries $\{v_1, v_2, \ldots, v_{2^q-1}\}$ and its corresponding multi-query graph $G = (V, E)$.
Output: a program that computes the expressions represented by the multi-query graph.
Method:
Repeat choosing the lowest numbered option among the following set of options.

1. Instantiate whenever possible. If a relation becomes empty, then eliminate all graphs connected to this node.

2. Iterate on the relation that provides the highest cost saving. That is, iterate on the relation represented by node r such that $savings(r) = max\{savings(r_i) \mid 1 \leq i \leq p\} > 0$.

   If there is a tie between nodes r and r', then iterate on node r if

   $$\sum_{s \in adjacent(r)} \nu(r, s) \geq \sum_{t \in adjacent(r')} \nu(r', t),$$

   otherwise iterate on r'. Notice that with this tie breaker we are favoring the node that will lead to a faster dissection of the multi-query graph. As a consequence, we are choosing to iterate on a relation whose access will be shared by the largest number of queries.

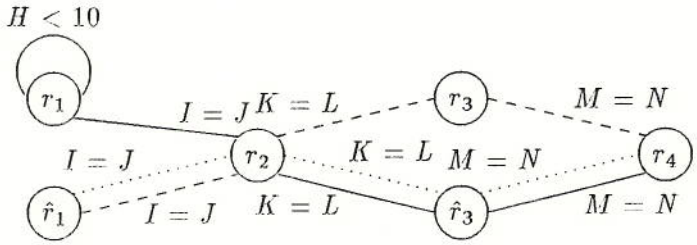
3. The algorithm reaches this point when $savings(r) = 0, \forall r \in V$. However, a node may still be connected to other nodes through single edges of several colors. Iterate on the node connected to the largest number of edges of different colors. If there are ties, then iterate on the node representing the relation with smallest cardinality.

4. When the algorithm reaches this point, the remaining portion of the graph consists of a number of disconnected components where each component contains edges of only one color. That means there are no more common subexpressions to be shared. Therefore, for $j = 1$ to $2^q - 1$: Find an optimal nesting order for computing the joins represented by the remaining edges in query j which are defined by the set $\{e \mid e = (r, r', j, \rho) \in E\}$ and then iterate on the relations for the component j according to the optimal nesting order.

Until the multi-query graph has no edges. □

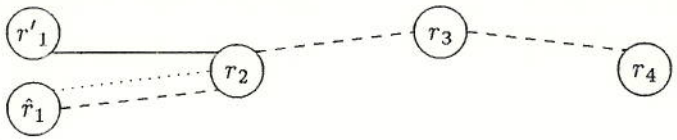
Algorithm H is applicable not only to the optimization of a set of queries in ΔV but to the optimization of an arbitrary set of queries. Example 5.1 illustrates Algorithm H.

Example 5.1 Consider the queries in Example 2.1: $v_1 = \sigma_{(H<10)\wedge(I=J)\wedge(K=L)\wedge(M=N)}(r_1 \times r_2 \times \hat{r}_3 \times r_4)$, $v_2 = \sigma_{(I=J)\wedge(K=L)\wedge(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$, and $v_3 = \sigma_{(I=J)\wedge(K=L)\wedge(M=N)}(\hat{r}_1 \times r_2 \times \hat{r}_3 \times r_4)$, with their corresponding multi-query graph:



where the expression v_1 is represented by solid edges, expression v_2 by dashed edges, and expression v_3 by dotted edges. We assume the same cardinalities of relations and selectivities of predicates as in Example 2.1.

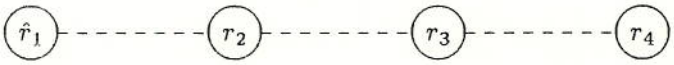
In the first iteration of the algorithm, the action taken is to instantiate relation r_1 (Step 1). This action removes the edge $(r_1, r_1, 1, (H < 10))$ producing a new node r_1' replacing node r_1. The code generated by this step is: $r_1' \leftarrow \sigma_{(H<10)}(r_1)$. The size of relation r_1' is estimated to be $scl_{11} * |r_1| = (0.1)(1000) = 100$ pages. In the second iteration of the algorithm no more instantiations are possible, therefore Step 2 is performed. In this step, the function *savings* is computed for each of the nodes of the graph. The values of this function for each of the relations is: $savings(r_1') = 0$, $savings(\hat{r}_1) = 88$, $savings(r_2) = 272$, $savings(r_3) = 0$, $savings(\hat{r}_3) = 596$, and $savings(r_4) = 412$. Since $savings(\hat{r}_3)$ gives the largest value, the next action taken by the algorithm is to iterate on $\hat{r}_3$. This action removes the edges $(r_2, \hat{r}_3, 1, (K = L))$, $(r_2, \hat{r}_3, 3, (K = L))$, $(\hat{r}_3, r_4, 1, (M = N))$, and $(\hat{r}_3, r_4, 3, (M = N))$. The resulting graph is:



The code generated by this step is:

$r_1' \leftarrow \sigma_{(H<10)}(r_1)$;
$v_1 \leftarrow \emptyset$; $v_3 \leftarrow \emptyset$;
for each t **in** $\hat{r}_3$ **do**
  $v_1^2 \leftarrow \sigma_{(I=J)(K=t[L])(t[M]=N)}(r_1' \times r_2 \times r_4)$;
  $v_1 \leftarrow v_1 \cup (v_1^2 \times \{t\})$;
  $v_3^2 \leftarrow \sigma_{(I=J)(K=t[L])(t[M]=N)}(\hat{r}_1 \times r_2 \times r_4)$;
  $v_3 \leftarrow v_3 \cup (v_3^2 \times \{t\})$;
od;
$v_2 \leftarrow \sigma_{(I=J)(K=L)(M=N)}(\hat{r}_1 \times r_2 \times r_3 \times r_4)$;

At the seventh iteration, the multi-query graph has a connected component with single edges all of the same color (i.e., a query graph for a single query), shown below:



At this point, Algorithm H switches to an optimization strategy better than single-query decomposition. This instance of the algorithm uses a routine that finds an optimal nesting order for computing the expression represented by the connected component of the graph [9,11]. The nesting order obtained is then used to decompose the rest of the graph. If at this stage in the algorithm, the multi-query graph contains several connected components each belonging to a single query, then single-query optimization (as described above) will be applied to each component. For this example, the optimal nesting order is given by the expression $(\hat{r}_1 \times (r_2 \times r_3 \times r_4))$. The complete code for the set of queries is shown below:

	Read	Write
1. $r_1' \leftarrow \sigma_{(H<10)}(r_1)$;	$cost = 1000$	$\|r_1'\| = 1000 * 0.1 = 100$
2. $v_1 \leftarrow \emptyset$; $v_3 \leftarrow \emptyset$;		
3. **for each** t **in** $\hat{r}_3$ **do**		
4. $r_2' \leftarrow \sigma_{(K=t[L])}(r_2)$;	$cost = 80$	$\|r_2'\| = 80 * 0.15 = 12$
5. $r_4' \leftarrow \sigma_{(t[M]=N)}(r_4)$;	$cost = 200$	$\|r_4'\| = 200 * 0.03 = 6$
6. $v_1^2 \leftarrow \emptyset$; $v_3^2 \leftarrow \emptyset$;		
7. **for each** s **in** r_2' **do**		
8. $r_1'' \leftarrow \sigma_{(I=s[J])}(r_1')$;	$cost = 100$	$\|r_1''\| = 100 * 0.1 = 10$
9. $\hat{r}_1' \leftarrow \sigma_{(I=s[J])}(\hat{r}_1)$;	$cost = 1$	$\|\hat{r}_1'\| = 1$
10. $v_1^4 \leftarrow (r_1'' \times r_4')$;	$cost = 60$	$\|v_1^4\| = 60$
11. $v_1^2 \leftarrow v_1^2 \cup (v_1^4 \times \{s\})$;	$cost = 60$	$\|v_1^2\| = 60$
12. $v_3^4 \leftarrow (\hat{r}_1' \times r_4')$;	$cost = 6$	$\|v_3^4\| = 6$
13. $v_3^2 \leftarrow v_3^2 \cup (v_3^4 \times \{s\})$;	$cost = 6$	$\|v_3^2\| = 6$
14. **od**;	$cost = (110 + 2 + 120 + 120 + 36) * 12 = 4,656$	
15. $v_1 \leftarrow v_1 \cup (v_1^2 \times \{t\})$;		
16. $v_3 \leftarrow v_3 \cup (v_3^2 \times \{t\})$;		
17. **od**;	$cost = (92 + 206 + 4,656) * 2 = 9,908$	
18. $v_2 \leftarrow \emptyset$;		
19. **for each** u **in** $\hat{r}_1$ **do**		
20. $r_2'' \leftarrow \sigma_{(u[I]=J)}(r_2)$;	$cost = 80$	$\|r_2''\| = 80 * 0.1 = 8$
21. $v_2^7 \leftarrow \emptyset$;		
22. **for each** w **in** r_3 **do**		
23. $r_2''' \leftarrow \sigma_{(K=w[L])}(r_2'')$;	$cost = 8$	$\|r_2'''\| = 8 * 0.15 = 1.2 \approx 2$
24. $r_4' \leftarrow \sigma_{(w[M]=N)}(r_4)$;	$cost = 200$	$\|r_4'\| = 200 * 0.03 = 6$
25. $v_2^{11} \leftarrow (r_2''' \times r_4')$;	$cost = 12$	$\|v_2^{11}\| = 12$
26. $v_2^7 \leftarrow v_2^7 \cup (v_2^{11} \times \{w\})$;	$cost = 12$	$\|v_2^7\| = 12$
27. **od**;	$cost = (10 + 206 + 24) * 90 = 21,600$	
28. $v_2 \leftarrow v_2 \cup (v_2^7 \times \{u\})$;	$cost = 12$	$\|v_2\| = 12$
29. **od**	$cost = (88 + 24 + 21,600) * 1 = 21,712$;	

The cost of the strategy for computing the set of queries v_1, v_2, and v_3 produced by Algorithm H

is: $cost = 1,100 + 9,908 + 21,712 = 32,720$. In contrast, the cost of performing each of the queries independently is 18,456 for v_1, 23,164 for v_2 and 611 for v_3 for a total of 42,231. Therefore, using decomposition we obtain a 22.5% saving for the set of queries (with the given predicate selectivities and cardinalities of relations) in this example. □

5.2 Remarks

The results obtained in Example 5.1 deserve some discussion. Why has the hybrid approach yielded better results than the state-space search approach? One reason is that the hybrid approach allows the execution of multi-way joins (not only two-way joins). For example, at Step 3 of the program above, each accessed page from $\hat{r}_3$ is immediately joined with relations r_2 and r_4 in Steps 4 and 5. Similarly for Steps, 22, 23, and 24. Thus, there is a richer set of elementary tasks considered by the hybrid approach. Also, the hybrid approach provides many opportunities for pipelining intermediate results (although we have not reflected this in our cost estimations). We believe the optimization cost incurred by the hybrid approach will in general not be higher than that of the state-space search approach. Additional experimental work is needed to confirm our belief.

6 Conclusions

In this paper, we have formulated the problem of computing the set of changes required to incrementally update a materialized view as a special case of multiple-query optimization. We report our initial results on applying three mqo approaches to this problem.

First, in the simple approach we have proven that for our special mqo problem we do not need to test subsumption among tasks. This result provides a substantial saving on the optimization cost because we do not need to invoke a theorem prover to perform such a test. To prove this we have shown that it is very important to understand various properties about the behavior of the single-query optimizer. Unlike other work in this area, we have shown that the simple algorithm we propose is correct; our proof sheds some light on how the assumptions about plan generators affect the difficulty of establishing the correctness of even simple mqo algorithms. The simple approach can be integrated easily with a conventional single-query optimizer and can be used more generally to merge equivalent subexpressions among queries whose plans are represented by trees. Second, in the state-space search approach, we proposed a judicious way of generating plans exploiting the special structure of our problem. Third, in the hybrid approach we proposed an improved heuristic for multiple-query decomposition. We also show how a single-query optimizer (not necessarily based on query-decomposition) can be integrated with multiple-query decomposition. This is an

important step towards integrating seemingly disparate optimizers. Additional experimental work is underway to obtain more definitive answers on the best approach for our special mqo problem. Also, more work is needed to evaluate the trade-offs between the cost of optimization and the quality of the optimization for the computation of ΔV.

References

[1] Michel Adiba and Bruce G. Lindsay. "Database Snapshots." In *Proc. of the 6th. International Conference on Very Large Databases*, pages 86–91, Montreal (1980).

[2] Rafael Alonso, Daniel Barbara, Hector Garcia-Molina, and Soraya Abad. "Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems." In *Proc. of the 1988 International Conference on Extending Database Technology*, March 14–18, 1988, Venice, Italy.

[3] José A. Blakeley, Per-Åke Larson, Frank Wm. Tompa. "Efficiently Updating Materialized Views." In *Proc. of the ACM SIGMOD International Conference on Management of Data,*, pages 61–71, Washington, D.C. (May 1986).

[4] Chakravarthy, Upen S. and Jack Minker, "Multiple Query Processing in Deductive Databases." In *Proc. of the 12th. International Conference on Very Large Data Bases*, pages 384–391, Kyoto (August 1986).

[5] Umeshwar Dayal et al. *HiPAC: A Research Project in Active, Time-constrained Database Management.* Technical Report CCA-88-02, Computer Corporation of America (June 1988).

[6] Sheldon Finkelstein. "Common Expression Analysis in Database Applications." In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 235–245, Orlando, FL. (June 1982).

[7] John Grant and Jack Minker. "On Optimizing the Evaluation of a Set of Expressions." *International Journal of Computer and Information Sciences.* Vol. 11, No. 3, pages 179–191 (June 1982).

[8] Eric Hanson. "A Performance Analysis of View Materialization Strategies." In *Proc. ACM SIGMOD International Conference on Management of Data* (1987).

[9] Toshihide Ibaraki and Tiko Kameda. "On the Optimal Nesting Order for Computing N-Relational Joins." *ACM Transactions on Database Systems*, Vol. 9, No. 3, pages 482–502 (September 1984).

[10] Matthias Jarke. "Common Subexpression Isolation in Multiple Query Optimization." In *Query Processing in Database Systems*, W. Kim, D. Reiner, D. Batory (eds.), Springer-Verlag, pages 191–205 (1985).

[11] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. "Optimization of Nonrecursive Queries." In *Proc. of the 12th. International Conference on Very Large Data Bases*, pages 128–137, Kyoto (1986).

[12] Per-Åke Larson and H. Z. Yang. "Computing Queries from Derived Relations." In *Proc. of the 11th International Conference on Very Large Data Bases*, pages 259–269, Stockholm (1985).

[13] Bruce Lindsay, Laura Hass, C. Mohan, Hamid Pirahesh, and Paul Wilms. "A Snapshot Differential Refresh Algorithm." In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 53–60, Washington, D.C. (1986).

[14] David Maier. *The Theory of Relational Databases.* Computer Science Press (1983).

[15] Jooseok Park and Arie Segev. "Using Common Subexpressions to Optimize Multiple Queries." In *Proc. of the 4th International Conference on Data Engineering,* pages 311–319 (1988).

[16] Tore Risch, René Reboh, Peter Hart, and Richard Duda. "A Functional Approach to Integrating Database and Expert Systems." *Communications of the ACM,* Vol. 31, No. 12, 1424–1437 (December 1988).

[17] Arnon Rosenthal and Upen S. Chakravarthy. "The Anatomy of a Multiple Query Optimizer." In *Proc. of VLDB,* pages 210-215 (1988).

[18] Nicholas Roussopoulos and Hyunchul Kang. "Preliminary Design of ADMS±: A Workstation-Mainframe Integrated Architecture for Database Management Systems." In *Proceedings of the 12th International Conference on Very Large Data Bases,* pages 355–364, Kyoto (August 1986).

[19] Arie Segev and Jooseok Park. *Updating Distributed Materialized Views.* Laurence Berkeley Laboratory, LBL-24882 (August 1988).

[20] 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 "Proc. of the ACM SIGMOD 1979 International Conference on Management of Data," pages 23–34 (1979).

[21] Timos K. Sellis. "Multiple Query Optimization." In *ACM Transactions on Database Systems,* Vol. 13, No. 1, pages 23–52, March 1988.

[22] M. Stonebraker, J. Anton, and E. Hanson. "Extending A Database System with Procedures." *ACM Transactions on Database Systems,* Vol. 12, No. 3, pages 350–376 (September 1987).

[23] Dionysios C. Tsichritzis and Frederick H. Lochovsky. *Data Base Management Systems.* Academic Press, (1977).

[24] Jeffrey D. Ullman. *Principles of Database Systems,* Computer Science Press, 2nd. edition (1982).

[25] Eugene Wong and Karel Youssefi. "Decomposition - A Strategy for Query Processing." *ACM Transactions on Database Systems,* Vol. 1, No. 3, pages 223–241 (September 1976).