Multiple-Query Optimization

TIMOS K. SELLIS
University of California, Berkeley

Some recently proposed extensions to relational database systems, as well as to deductive database systems, require support for multiple-query processing. For example, in a database system enhanced with inference capabilities, a simple query involving a rule with multiple definitions may expand to more than one actual query that has to be run over the database. It is an interesting problem then to come up with algorithms that process these queries together instead of one query at a time. The main motivation for performing such an interquery optimization lies in the fact that queries may share common data. We examine the problem of multiple-query optimization in this paper. The first major contribution of the paper is a systematic look at the problem, along with the presentation and analysis of algorithms that can be used for multiple-query optimization. The second contribution lies in the presentation of experimental results. Our results show that using multiple-query processing algorithms may reduce execution cost considerably.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General; H.2.4 [**Database Management**]: Systems—*query processing*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*heuristic methods*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Common access paths, deductive databases, query optimization, relational databases, sharing of data

1. INTRODUCTION

In the past few years, several attempts have been made to extend the benefits of the database approach in business to other areas, such as artificial intelligence and engineering design automation. As a result, various extensions to database query languages have been suggested, including QUEL* [18], designed to support artificial intelligence applications; GEM [31], to support a semantic data model; and the proposal of [11], for support of VLSI design databases. A significant part of extended database languages is support for multiple command processing. In [26] we proposed a set of transformations and tactics for optimizing collections of commands in the presence of updates. Here, we will concentrate on the problem of optimizing the execution of a set of retrieve-only commands (queries).

This research was sponsored by the U.S. Air Force Office of Scientific Research grant 83-0254 and by the National Science Foundation under grant DMC-8504633.
Author's address: Department of Computer Science, University of Maryland, College Park, MD, 20742.

There are many applications where more than one query is presented to the system in order to be processed. First, consider a database system enhanced with inference capabilities (*deductive database system*) [8]. A single query given to such a system may result in multiple queries that will have to be run over the database. As an example, consider the following relation for employees EMP(name, salary, experience, dept-name). Assume also the existence of a set of rules that define when an employee is well paid. We express these rules in terms of retrieve commands in QUEL [28].

/* *An employee is well paid if he(she) makes more than* 40*K* */
Rule 1. retrieve (EMP.all) where EMP.salary > 40
/* *An employee is well paid if he(she) makes more than* 35*K provided he(she) has no more than* 5 *years of experience* */
Rule 2. retrieve (EMP.all) where EMP.salary > 35 and EMP.experience ≤ 5
/* *An employee is well paid if he(she) makes more than* 30*K provided he(she) has no more than* 3 *years of experience* */
Rule 3: retrieve (EMP.all) where EMP.salary > 30 and EMP.experience ≤ 3

Then, given a query that asks

Is Mike well paid?

the system will have to evaluate all three rules in order to come up with the answer. Because of the similarities that Prolog [6] clauses have with the above type of rules, our discussion on multiple-query processing applies to the optimization of Prolog programs as well, assuming that secondary storage is used to hold a Prolog database of facts. As a second example, consider cases where queries are given to the system from various users. Then *batching* all users' requests is a possible processing strategy. In particular, queries given within the same time interval τ may be considered for batched processing. However, a major problem with this approach is the effect on response time. It is unacceptable to delay a user's request due to other more expensive queries. Although it is a very interesting problem to find criteria for batching multiple requests, we will gear the discussion toward a system like the rule-based system mentioned above, where a single user request is expanded to many actual queries. Finally, some proposals on processing recursion in database systems [14, 20] suggest that a recursive Horn clause should be transformed to a set of other, simpler Horn clauses (recursive and nonrecursive). Therefore, the problem of multiple-query processing arises in that environment as well, yet in a more complicated form due to the presence of recursion.

Current query processors cannot optimize the execution of more than one query. If given a set of queries, the common practice is to process each query separately. However, there may be some common tasks that are found in more than one of these queries. Examples of such tasks may be performing the same restriction on a relation or performing the same join between two relations. Taking advantage of these common tasks, mainly by avoiding redundant page accesses, may prove to have a considerable effect on execution time. This problem of processing multiple queries and especially the optimization of their execution, will be the focus of this paper. Section 2 presents an overview of previous work

in the area. Section 3 first defines the query model that will be used throughout this paper and then presents a formulation of the multiple-query optimization problem. Section 4 presents our approach to the problem and introduces, through the use of some examples, algorithms that can be used to solve the multiple-query optimization problem. Then, Sections 5 and 6 present these algorithms in more detail. Section 5 suggests an algorithm that allows the executions of the queries to interleave, thus improving the performance compared to a serial execution, and Section 6 discusses a more general heuristic algorithm. Finally, in Section 7 we present some experimental results, and the last section concludes the presentation of the multiple-query processing problem by summarizing our results and suggesting some ideas for future research.

2. RELATED WORK

Problems similar to the multiple-query processing problem have been examined in the past in various contexts. Hall [12, 13], for example, uses heuristics to identify common subexpressions, especially within a single query. He uses operator trees to represent the queries and a bottom-up traversal procedure to identify common parts. In [9] and [10], Grant and Minker describe the optimization of sets of queries in the context of deductive databases and propose a two-stage optimization procedure. During the first stage ("Preprocessor"), the system obtains at *compile* time information on the access structures that can be used in order to evaluate the queries. Then, at the second stage, the "Optimizer" groups queries and executes them in groups instead of one at a time. During that stage common tasks are identified and sharing the results of such tasks is used to reduce processing time. Roussopoulos, in [24] and [25], provides a framework for interquery analysis based on query graphs [30], in an attempt to find fast access paths for view processing ("view indexing"). The objective of his analysis is to identify all possible ways to produce the result of a view, given other view definitions and base relations. Indexes are then built as data structures to support fast processing of views.

Other researchers have also recently examined the problem of multiple-query optimization. Chakravarthy and Minker [3, 4] propose an algorithm based on the construction of integrated query graphs. These graphs are extensions of the query graphs introduced by Wong and Youssefi in [30]. Using integrated query graphs, Chakravarthy and Minker suggest a generalization of the query decomposition algorithm of [30]; however, this algorithm does not guarantee that the access plan constructed is the cheapest one possible. Kim, in [17], also suggests a two-stage optimization procedure similar to the one in [10]. The unit of sharing among queries in Kim's proposal is the relation that is not always the best thing to assume, except in cases of single relation queries.

The work of [7] and [19] on the problem of deriving query results based on the results of other previously executed queries is also related to the problem of multiple-query optimization. Finally, Jarke discusses in [16] the problem of common subexpression isolation. He presents several different formulations of the problem under various query language frameworks such as relational algebra, tuple calculus, and relational calculus. In the same paper, he also describes how common expressions can be detected and used according to their type (e.g., single relation restrictions, joins, etc).

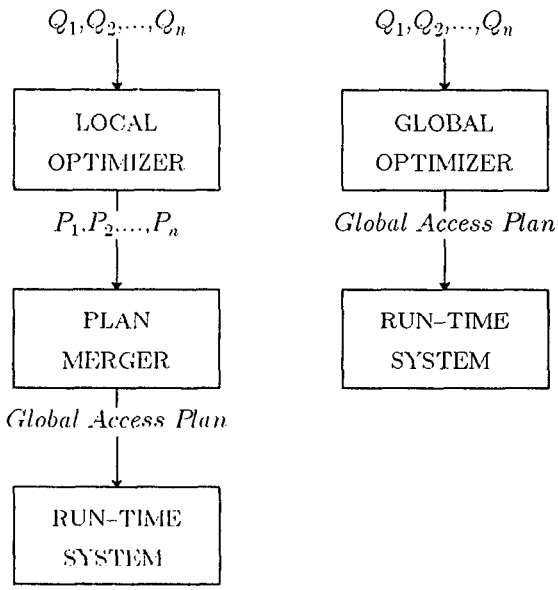

Fig. 1. Multiple-query processing systems architecture.

What distinguishes our approach to multiple-query processing is the decision to use existing query optimizers as much as possible. However, since not all relational database systems have been designed on the basis of the same query processing concepts, we will differentiate between two alternative architectures that can be used for a system with multiple-query processing capability. Figure 1 illustrates the two approaches. Architecture 1 can be used with minimal changes to existing optimizers. A conventional *Local Optimizer* generates one ("locally") optimal access plan per query. The *Plan Merger* is a component that examines all access plans and generates a larger plan, the "global" access plan, which is in turn processed by the *Run-Time System*. This architecture is particularly interesting for systems that *compile* queries and save results in the form of access plans (e.g., System-R [1], POSTGRES [27]).

On the other hand, there are systems that do not store access plans for future reuse (e.g., INGRES [28]). To make our framework general enough to capture these systems as well, we introduce Architecture 2. The set of queries is processed by a more sophisticated component, the *Global Optimizer*, which in turn passes the derived global access plan to the Run-Time System for processing. Hence, Architecture 2 is not restricted to solely using locally optimal plans already stored in the system. Notice also that this architecture can be used for the development of a multiple-query optimization module from scratch (for example, the optimizer for a deductive database system [10]).

The purpose of the following sections is to exhibit optimization algorithms that can be used for multiple-query optimization either as Plan Mergers or as Global Optimizers. The algorithms to be presented differ in the complexity of the Plan Merger and on whether Architecture 1 or 2 is used. The tradeoffs between the complexity of the algorithms and the optimality of the global plan produced are also discussed.

3. FORMULATION OF THE PROBLEM

We assume that a *database* **D** is given as a set of *relations* $\{R_1, R_2, \ldots, R_m\}$, each relation defined on a set of *attributes*. A simple model for queries is now described. A *selection predicate* is a predicate of the form $R.A$ *op cons*, where R is a relation, A an attribute of R, *op* $\in \{=, \neq, <, \leq, >, \geq\}$, and *cons* some constant. A *join predicate* is a predicate of the form $R_1.A = R_2.B$ where R_1 and R_2 are relations, and A and B are attributes of R_1 and R_2, respectively (*equijoin*). For simplicity we will assume that the given queries are conjunctions of selection and join predicates and *all* attributes are returned as the result of the query (i.e., we assume no *projection* on specific attributes). Clearly, the above model excludes aggregate computations or functions as well as predicates of the form $R_1.A$ *op* $R_2.B = R_3.C$. Extending a system to support such predicates is possible but would require significant increase to its complexity. The restriction on conjunctive queries only is not a severe limitation since the result of a disjunctive query can be considered as the union of the results of the disjuncts, i.e., each disjunct can be thought of as a different query. Equijoins are chosen as the only join operator; this seems quite natural considering the most common types of queries. Finally, not allowing projections enables us to concentrate on the problem of sharing common results rather than the problem of detecting if the result of a query can be used to compute the result of another query. However, had we assumed projection lists as well, the complexity of the algorithms that detect results which can be shared among queries would be higher (see [7] and [19] for such algorithms).

A *task* is an expression *relname* $\leftarrow$ *expr*. *relname* is the name of a temporary relation used to store an intermediate result or the keyword *RESULT*, indicating that this task provides the result of the query. *expr* is either a conjunction of selection predicates over the same relation or a conjunction of joins between the same two, possibly restricted, relations. For example, the following are valid task expressions:

E1: $R_1.A = 10$ and $R_1.C \leq 30$
E2: $R_1.A = R_2.B$ and $R_1.C = R_2.D$
E3: $(R_1.A = 10).C = (R_2.B < 30).D$

The cases of joins like those in *E3* cover queries that are processed in a "pipelining" way, not by performing the selections first followed by a join. For example, one way to process *E3* is by scannning the relation R_1 and having each tuple with qualifying A value be checked against R_2 tuples. There is no need to store intermediate results for either R_1 or R_2. Our model is general enough to include this kind of processing as well. In the remaining discussion, tasks will be referred to as if they were simply the *expr* part, unless otherwise explicitly stated. We next define a partial order on tasks.

Definition 1. A task t_i *implies* task t_j ($t_i \Rightarrow t_j$) iff t_i is a conjunction of selection predicates on attributes $A_1, A_2, \ldots, A_k$ of some relation R, and t_j is a conjunction of selection predicates on the same relation R and on attributes $A_1, A_2, \ldots, A_l$ with $l \leq k$, and it is the case that for any instance of the relation R the result of evaluating t_i is a subset of the result of evaluating t_j.

Definition 2. A task t_i is *identical* to task $t_j (t_i \equiv t_j)$ iff

(a) *Selections*: $t_i \Rightarrow t_j$ and $t_j \Rightarrow t_i$

(b) *Joins*: t_i is a conjunction of join predicates $E_1.A_1 = E_2.B_1$, $E_1.A_2 = E_2.B_2$, ..., $E_1.A_k = E_2.B_k$ and t_j is a conjunction of join predicates $E_1'.A_1 = E_2'.B_1$, $E_1'.A_2 = E_2'.B_2$, ..., $E_1'.A_k = E_2'.B_k$ where each of E_1, E_2, E_1' and E_2' is a conjunction of selections on a single relation and E_1 is identical to E_1' and E_2 is identical to E_2' ("identical" under the above definition of identical selections).

Based on the above definitions, we will use the phrase "common subexpressions" to describe pairs of tasks t_1 and t_2 where either one implies the other or they are identical. Next, we define the notion of an access plan.

Definition 3. An *access plan* for a query Q is a sequence of tasks that produces the answer to Q. Formally, an access plan is an acyclic directed graph $P = (V, A, L)$ (V, A, and L being the sets of vertices, arcs, and vertex labels, respectively) defined as follows:

—For every task t_i of the plan introduce a vertex v_i.

—If the result of a task t_i is used in task t_j, introduce an arc $v_i \rightarrow v_j$ between the vertices v_i and v_j that correspond to t_i and t_j, respectively.

—The label $L(v_i)$ of vertex v_i is the processing done by the corresponding task t_i (i.e., *relname* ← *expr*).

Example 1. Consider the following query on the relations EMP(name, age, dept-name) and DEPT(dept-name, num-of-emps) (with obvious meanings for the various fields)

```
retrieve (EMP.all, DEPT.all)
  where EMP.age ≤ 40
  and DEPT.num-of-emps ≤ 20
  and EMP.dept-name = DEPT.dept-name
```

One way to process this query is

```
 TEMP1 ← EMP.age ≤ 40
 TEMP2 ← DEPT.num-of-emps ≤ 20
RESULT ← TEMP1.dept-name = TEMP2.dept-name
```

The graph of Figure 2 shows the corresponding access plan. Notice that, in general, there may exist many possible plans that can be used to process a query.

Next we define a cost function $cost: V \rightarrow \mathbb{Z}$ ($\mathbb{Z}$ is the set of integers) on nodes of the access plan graph. In general this cost depends on both the CPU time and the number of disk page accesses needed to process the given task. However, to simplify the analysis, we will consider only I/O costs; including CPU costs would only make the formulas more complex. Therefore,

$cost(v_i)$ = the number of page accesses (reads or writes) needed to process task t_i

The cost $Cost(P)$ of an access plan P is defined as

$$Cost(P) = \sum_{v_i \in V} cost(v_i) \tag{1}$$

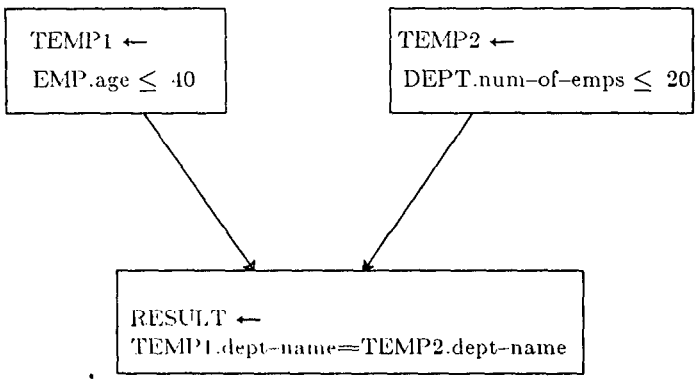

Fig. 2. Example of an access plan.

Assume now that a set of queries $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ is given. We will refer to the minimal cost plans for processing each query Q_i individually, as *locally optimal* plans. Similarly, we use the term *globally optimal* plan to refer to an access plan that provides a way to compute the results of all n queries with minimal cost. Due to common subexpressions, the union of the locally optimal plans is, in general, different from the globally optimal plan. Finally, let *Bestcost* be a function that given a query Q_i gives the cost of the (locally) optimal plan P_i^*. Hence, $Bestcost(Q_i) = Cost(P_i^*) = \min_{P_i \in \mathcal{P}_i}[Cost(P_i)]$, where $\mathcal{P}_i$ is the set of all possible plans that can be used to evaluate Q_i.

Consider now a system that is given a set $\mathcal{Q}$ of queries and is required to execute them with minimal cost. According to the above definitions, a global access plan is simply a directed labeled graph that provides a way to compute the results of *all* n queries. Based on this formulation, the problem of multiple-query optimization becomes

Given *n sets of access plans $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$, with $\mathcal{P}_i = \{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$ being the set of possible plans for processing Q_i, $1 \leq i \leq n$,*

Find *a global access plan GP by "merging" n local access plans (one out of each set $\mathcal{P}_i$) such that $Cost(GP)$ is minimal.*

The Plan Merger or the Global Optimizer of Figure 1 performs the "merging" operation mentioned above. It is the purpose of the following sections to define this operation and derive algorithms that find *GP*.

4. MOTIVATION FOR ALGORITHMS

The major issue in multiple-query processing is the redundancy due to accessing the same data multiple times in different queries. Recognizing all possible cases where the same data is accessed multiple times requires, in general, a procedure equivalent to theorem proving, including the retrieval of data from the database. Our intention here is to detect common subexpressions looking only at the logical expressions used to describe queries, that is, by simply isolating pairs of expressions e_1 and e_2 where $e_1 \Rightarrow e_2$. For example, e_1 may be EMP.age $\leq$ 30 and e_2 may be EMP.age $\leq$ 40. Then $e_1 \Rightarrow e_2$. However, we do not consider cases where

e_2 may be EMP.dept-name = "shoe", and it happens in the specific instance of the database that all employees under 40 years old work in the shoe department. Unless such a rule is explicitly known to the system in the form of an integrity constraint or functional dependency, it is not possible to detect that $e_1 \Rightarrow e_2$ without looking at the actual data stored [2, 5, 15]. Because several algorithms have been published in the past on the problem of common subexpression isolation [7, 19, 23], we will not attempt here to present a similar algorithm. It is assumed that a procedure that decides, given two expressions e_1 and e_2, if $e_1 \Rightarrow e_2$ or $e_2 \Rightarrow e_1$, is available.

A global access plan that is derived based on the idea of temporary result sharing should be less expensive compared to a serial execution of queries. However, this cannot be true for *any* database state. For example, sharing temporary results may prove to be a bad decision when indexes on relations are defined. The cost of processing a selection through an index or through an existing temporary result clearly depends on the size of these two structures. The experimental results of Section 7 give some interesting results regarding that issue. In general, a multiple-query optimization strategy should be compared to a conventional one, where no sharing is assumed, and the cheapest one should be selected (Finkelstein makes a similar argument in [7]). The conventional strategy will be computed one way or the other since, as mentioned above, locally optimal plans for the queries are always available.

In this paper we will examine two types of algorithms that agree with the two types of architectures shown in Figure 1. The first two algorithms consider only access plans that are *locally* optimal. Algorithm **AS** (Arbitrary Serial Execution) simply executes these plans in an arbitrary order (conventional approach). This corresponds to Architecture 1 of Figure 1 with the Plan Merger absent, that is, no optimization is performed. We include **AS** in our discussion to be used solely as a reference for the rest of the algorithms. Algorithm **IE** (Interleaved Execution) allows queries to be decomposed into smaller subqueries that now become the unit of execution. Therefore, a query is not processed as a whole but rather in small pieces, the results of which are assembled at various points to produce the answer to the original query.

Example 2. To illustrate algorithm **IE**, consider the following database,

```
EMP(name, age, salary, job, dept-name)
DEPT(dept-name, num-of-emps)
JOB(job, project)
```

with the obvious meanings for EMP, DEPT, and JOB. We also assume that there are no fast access paths for any of the relations, and that the following queries

```
(Q1)  retrieve (EMP.all, DEPT.all)
        where EMP.age ≤ 40
        and DEPT.num-of-emps ≤ 20
        and EMP.dept-name = DEPT.dept-name

(Q2)  retrieve (EMP.all, DEPT.all)
        where EMP.age ≤ 50
        and DEPT.num-of-emps ≤ 10
        and EMP.dept-name = DEPT.dept-name
```

are given. Finally, suppose that both Q_1 and Q_2 have optimal plans that construct temporary results based on the constraints on age and num-of-emps. If we run either Q_1 or Q_2 first, we will be unable to use the intermediate results from the restrictions on EMP.age and DEPT.num-of-emps effectively. However, the following global access plan is more efficient (for clarity, hereafter, unless otherwise stated, we show the plans in terms of QUEL queries instead of directed graphs)

```
retrieve into tempEMP (EMP.all)
  where EMP.age ≤ 50
retrieve into tempDEPT (DEPT.all)
  where DEPT.num-of-emps ≤ 20
retrieve (tempEMP.all, tempDEPT.all)
  where tempEMP.age ≤ 40
  and tempEMP.dept-name = tempDEPT.dept-name
retrieve (tempEMP.all, tempDEPT.all)
  where tempDEPT.num-of-emps ≤ 10
  and tempEMP.dept-name = tempDEPT.dept-name
```

because it avoids accessing the EMP and DEPT relations more than once.

Algorithm **IE** can generate very efficient global access plans especially in cases where restrictions reduce the sizes of the original relations significantly. The function of the Plan Merger, in the case of algorithm **IE**, is to "glue" the plans together in a way that provides better utilization of common temporary (intermediate) results.

The second algorithm we present, algorithm **HA** (**H**euristic **A**lgorithm), is based on searching among local (not necessarily optimal) query plans and building a global access plan by choosing one local plan per query. Architecture 2 of Figure 1 applies to this case. The effectiveness of algorithm **HA** is illustrated with the following example.

Example 3. Suppose we are given the queries

```
(Q3)  retrieve (JOB.all, EMP.all, DEPT.all)
        where EMP.dept-name = DEPT.dept-name
        and JOB.job = EMP.job
(Q4)  retrieve (EMP.all, DEPT.all)
        where EMP.dept-name = DEPT.dept-name
```

to be processed over the database of Example 2. Assume also that Q_3 and Q_4 have optimal local plans

```
(P31)  retrieve into TEMP1 (JOB.all, EMP.all)
         where JOB.job = EMP.job
       retrieve (TEMP1.all, DEPT.all)
         where TEMP1.dept-name = DEPT.dept-name
(P41)  retrieve (EMP.all, DEPT.all)
         where EMP.dept-name = DEPT.dept-name
```

respectively. Notice that P_{31} and P_{41} do not share the common subexpression EMP.dept-name=DEPT.dept-name. Algorithm **HA** considers, in addition to P_{31},

the following plan for query Q_3

```
(P32)  retrieve into TEMP1 (EMP.all, DEPT.all)
         where EMP.dept-name = DEPT.dept-name
       retrieve (JOB.all, TEMP1.all)
         where JOB.job = TEMP1.job
```

Clearly, this allows the multiple-query optimization algorithm to consider more useful permutations of the plans.

In addition, **HA** uses some heuristics to reduce the number of permutations of plans it has to examine in order to find the optimal global plan.

We emphasize again the fact that algorithm **IE** works only on locally optimal plans and tries to achieve sharing based on these plans. Although that may not be the optimal strategy, we argue that given these plans, the algorithm suggested will make the best use of existing temporary results. If not many temporary results are created (e.g., in the "pipeline" way of processing a join), simply no sharing will be possible. Algorithms **IE** and **HA** are examined in more detail in the following two sections.

5. INTERLEAVED EXECUTION ALGORITHM

Since the sequence in which the queries are run is chosen arbitrarily in algorithm **AS**, the global plan GP that is produced is simply the concatenation in an arbitrary way of the locally optimal plans. Therefore, for any order of processing (execution) $\varepsilon = \{P^*_{i_1} P^*_{i_2} \cdots P^*_{i_n}\}$, with $i_k \in \{1, 2, \ldots, n\}$ and all i_k distinct, the cost of the global access plan will be

$$Cost(GP) = \sum_{i=1}^{n} Bestcost(Q_i) \tag{2}$$

As mentioned in the previous section, the basic idea behind algorithm **IE** is to allow the execution of various access plans to *interleave*. This is achieved by decomposing the given queries into smaller subqueries and running them in some order, depending on the various relationships among the queries. Then, the results of subqueries are assembled to generate the answers to the original queries. The only restriction imposed is that the partial order defined on the execution of tasks in a local access plan must be preserved in the global access plan as well.

Algorithm **IE** proceeds as follows. First, the queries that possibly overlap on some selections or joins are identified by checking the base relations that are used. For any query $Q_i \in \mathscr{Q}$ that overlaps with some other query, we consider the corresponding local access plan $P_i^*(V_i, A_i, L_i)$ and define a directed labeled graph GP(Global Access Plan) that represents the "union" of all such local plans. Formally, the graph $GP(GV, GA, GL)$ is defined as follows:

—$GV = \bigcup_{i=1}^{n} V_i$
—$GA = \bigcup_{i=1}^{n} A_i$
—For every $v_i \in V_i$, $GL(v_i) = L_i(v_i)$.

We will also assume that the result node of query Q_i contains the keyword $RESULT_i$ to indicate that this specific node provides the answer to that query. Based on this graph, the algorithm performs some simple steps that introduce

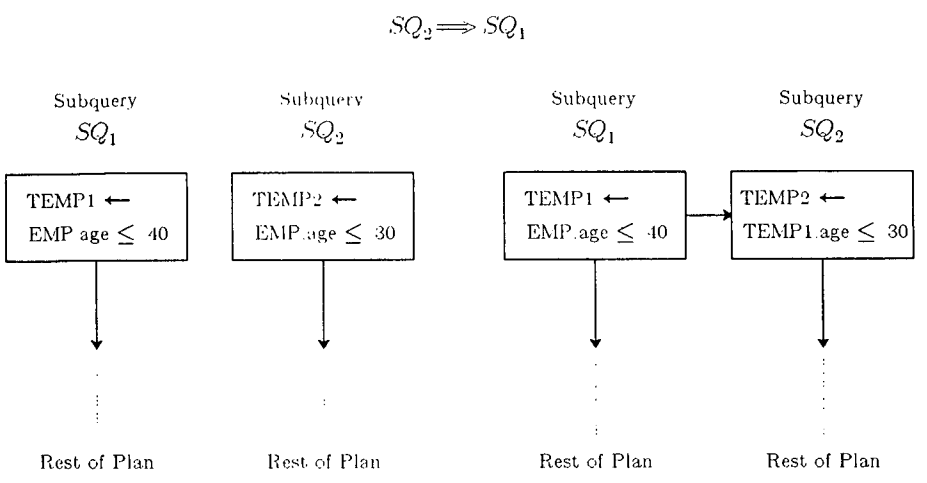

Fig. 3. Basic merge operation.

the effects of sharing among various tasks. Figure 3 illustrates the basic transformation. The temporary relation TEMP1 created by subquery SQ_1 can be further restricted to give the result of subquery SQ_2 ($SQ_2 \Rightarrow SQ_1$). Therefore, TEMP1 can be used as the input to that last subquery, instead of EMP. This is accomplished by adding a new arc from the node representing SQ_1 to the corresponding node for SQ_2. Also the relation name in SQ_2 is changed to TEMP1.

After building the graph *GP*, the following transformations are performed in the order they are presented

IE1. ⟨*Proper Implications*⟩ For a task v_i, let v_j be the nodes such that $GL(v_i) \Rightarrow GL(v_j)$ and $GL(v_j) \not\Rightarrow GL(v_i)$. We denote by v_{j^*} the *strongest* condition that can be performed on some of v_i's input relation(s) so that the result of v_{j^*} can still be used to answer v_i. By "strongest" we mean that v_{j^*}'s result is the smallest in terms of pages, among all such v_j's. Once the v_{j^*} nodes have been found, we apply the merge operation of Figure 3 on v_i to substitute input relations with the result of v_{j^*}

IE2. ⟨*Identical Nodes*⟩ In the case where there is a set C of nodes such that all its members produce identical temporary relations, we choose the one belonging to the plan P_j^* with the least index j as the representative node v_{j^*} of C. Then, as in step **IE1** we apply the merge operation of Figure 3 on all nodes $v_i \in C - \{v_{j^*}\}$ to substitute input relations with the result of v_{j^*}.

IE3. ⟨*Recursive Elimination*⟩ Because steps **IE1** and **IE2** may have introduced new nodes that are now identical, step **IE2** is repeatedly applied until it fails to produce any further reduction to the graph *GP*. An example of such a case is a join performed on two relations that are restricted with identical selection clauses. Step **IE2** will merge each pair of identical selections to a single one (by substituting temporary relation names); then, in the next iteration, the two join nodes will also be merged into a single node.

The result of the above transformation is a directed graph *GP'*, which is guaranteed to be acyclic if the initial graphs P_i^* are acyclic. This is due to the

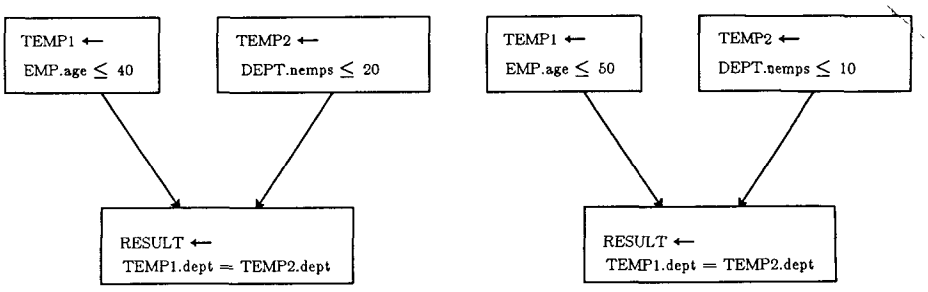

Fig. 4. Initial global access plan.

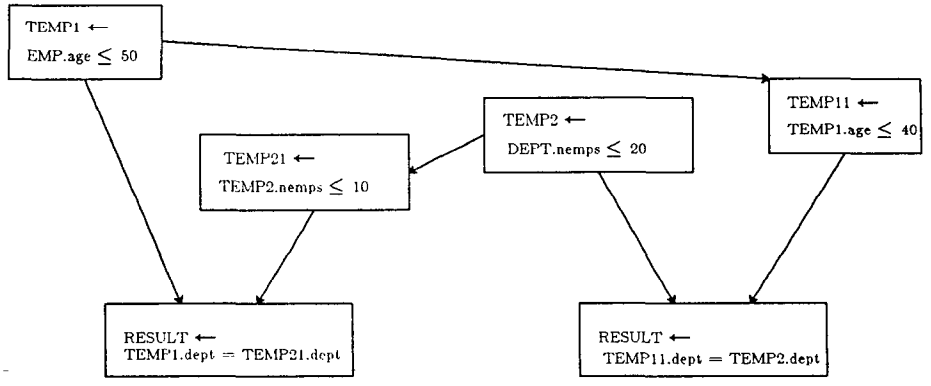

Fig. 5. Global access plan after transformation **IE1**.

fact that any transformation performed on the graph in all cases adds new arcs that go always from less to more restrictive tasks. Therefore, a cycle is not possible, for it would introduce a chain of proper implications of the form $v_1 \Rightarrow v_2 \Rightarrow \cdots \Rightarrow v_1$. Finally, using the directed arcs of GP' a partial order on the execution of the various tasks can be imposed. That is the global access plan that algorithm **IE** suggests.

Example 4. Consider again queries Q_1 and Q_2 of Example 2. Figures 4, 5, and 6 show the initial access plan graphs, the graph GP after transformation **IE1**, and the final global access plan graph (as a sequence of *QUEL* operations), respectively. (In Figures 4 and 5 we use nemps for num-of-emps and dept for dept-name).

Notice how in this case the algorithm makes use of the common subexpressions DEPT.num-of-emps ≤ 20 and EMP.age ≤ 50.

Estimating the cost of the global plan imposed by the graph GP', we have

$$Cost(GP') = \sum_{i=1}^{n} Bestcost(Q_i) - \sum_{s \in CS} savings(s) \tag{3}$$

where CS is the set of *all* (maximal) common subexpressions found in the local access plans and $savings(s)$ is the cost that is saved if the temporary result of a

```
retrieve into TEMP1 (EMP.all)
   where EMP.age ≤ 50
retrieve into TEMP2 (DEPT.all)
   where DEPT.num-of-emps ≤ 20
retrieve into TEMP11 (TEMP1.all)
   where TEMP1.age ≤ 40
retrieve into TEMP21 (TEMP2.all)
   where TEMP2.num-of-emps ≤ 10
retrieve (TEMP11.all,TEMP2.all)
   where TEMP11.dept-name = TEMP2.dept-name
retrieve (TEMP1.all,TEMP21.all)
   where TEMP1.dept-name = TEMP21.dept-name
```

Fig. 6. Final global access plan.

common subexpression s instead of base relations is used. In this example CS = {EMP.age≤50,DEPT.num-of-emps≤20}. The function *savings* is defined as follows:

Let R be a relation and s_1 and s_2 be two subexpressions defined on R such that s_2 can be processed using the result of s_1 instead of R. Let also C_R be the cost of accessing R to evaluate s_1 and C_{s_1} be the cost of accessing the result of s_1 to evaluate s_2. We assume that the results of s_1 and s_2 are stored for later use (temporary results). Then, without sharing any common results, the cost of processing s_1 is C_R(to read the data) + C_{s_1}(to write the result). The cost is similar for s_2. With sharing, the savings that can be achieved is

$$savings(s_2) = \begin{cases} C_R - C_{s_1} & \text{if } s_2 \Rightarrow s_1 \\ C_R + C_{s_1} & \text{if } s_2 \equiv s_1 \end{cases} \quad (4)$$

In the first case instead of accessing R we access the result of s_1, hence the savings of $C_R - C_{s_1}$. In the second case more savings are achieved because not only does R not need to be accessed (since the result of s_2 is identical to that of s_1), but the temporary result of s_1 can also be used as is as the result of s_2. Therefore, there is no need to write the result of s_2 in a separate temporary relation.

Concerning the complexity of the algorithm, it can be observed that steps **IE1** and **IE2** of the above algorithm require time in the order of $\Pi_{i=1}^{k} \mid V_i \mid$, where k is the number of queries represented by their representative plans in graph GP and V_i is the set of vertices for plans P_i^*, $1 \leq i \leq k$. The number of times N step **IE2** is executed as a result of the recursive elimination of common subgraphs generally depends on the size of common subexpressions and, in the worst case, is the depth of the longest query plan. The total time required by the algorithm is therefore in the order of $N \cdot \Pi_{i=1}^{k} \mid V_i \mid$.

We now move on to discuss a more general algorithm that can be used to process multiple queries. As mentioned in the beginning of this section, the heuristic algorithm to be described also captures more general transformations than the ones allowed here (simple relation name change).

6. HEURISTIC ALGORITHM

As it was illustrated through Example 3, merging locally optimal plans to produce the global access plan is not always the optimal strategy. The main reason is that there is more than one possible plan to process a query, yet algorithm **IE** considers only one of them, i.e., the locally optimal plan. Using suboptimal plans may prove to be better. Grant and Minker in [9] present a Branch and Bound algorithm [21] that uses more than locally optimal plans. One assumption they make is that queries involve only equijoins while all selections are of the form $R.A = cons$. In this section, we propose a general framework for the design of a heuristic multiple-query optimization algorithm. Then, we show how the algorithm of Grant and Minker can be mapped onto our more general algorithm, and we suggest some further improvement that aims to better performance. To simplify the presentation of the algorithm we will also make here the assumption that all queries have equality predicates. At the end of the section extensions that can be made to include more general query predicates are discussed.

As shown in Figure 1, the Global Optimizer receives as input a set of queries $\mathscr{Q} = \{Q_1, Q_2, \ldots, Q_n\}$. Then for each query Q_i, a set of possible plans $\mathscr{P}_i = \{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$ that can be used to process that query is derived. The algorithm **HA** considers optimizing a set of queries instead of a set of plans, which was the case with algorithm **IE**. Considering more than one candidate plan per query has the desirable effect of detecting and using effectively all common subexpressions found among the queries.

We will model the optimization problem as a state space search problem and propose the use of an A* algorithm [21]. In order to present an A* algorithm, one needs to define a state space, the way transitions are done between states and the costs of those transitions.

Definition 4. A *state* s is an n-tuple $\langle P_{1j_1}, P_{2j_2}, \ldots, P_{nj_n}\rangle$, where $P_{ij_i} \in \{\text{NULL}\} \cup \mathscr{P}_i$. If $P_{ij_i} = \text{NULL}$ it is assumed that state s suggests no plan for evaluating query Q_i. We denote $\mathscr{S}$ to be the set of all possible states.

Definition 5. Given a state $s = \langle P_{1j_1}, P_{2j_2}, \ldots, P_{nj_n}\rangle$, we define a function $next: \mathscr{S} \rightarrow \mathbb{Z}$ ($\mathbb{Z}$ is the set of integers) as follows

$$next(s) = \begin{cases} \min\{i \mid P_{ij_i} = \text{NULL}\} & \text{if } \{i \mid P_{ij_i} = \text{NULL}\} \neq \emptyset \\ n + 1 & \text{otherwise} \end{cases}$$

Let $s_1 = \langle P_{1j_1}, P_{2j_2}, \ldots, P_{nj_n}\rangle$ and $s_2 = \langle P_{1k_1}, P_{2k_2}, \ldots, P_{nk_n}\rangle$ be two states such that s_1 has at least one NULL entry. Also let $m = next(s_1)$. A *transition* $T(s_1, s_2)$ from state s_1 to state s_2 exists iff $P_{ik_i} = P_{ij_i}$, for $1 \leq i < m$, $P_{mk_m} \in \mathscr{P}_m$ and $P_{ik_i} = \text{NULL}$, for $m < i \leq n$.

Definition 6. The *cost* $tcost(t)$ of a transition $t = T(s_1, s_2)$ is defined as the *additional* cost needed to process the new plan P_{mk_m} introduced at t (according to Definition 5), given the (intermediate or final) results of processing the plans of s_1.

From the way transitions are defined, it is evident that the first NULL entry of a state vector, say at position i, will always be replaced by a plan for the corresponding query Q_i. Finally, we define the initial and final states for the

Table I. Costs for Tasks in Each Plan

Plan	Task	Cost	Task	Cost	Task	Cost	Total
P_{51}	t_{51}^1	40	t_{51}^2	30	t_{51}^3	5	75
P_{52}	t_{52}^1	35	t_{52}^2	20			55
P_{61}	t_{61}^1	40	t_{61}^2	10	t_{61}^3	5	55
P_{62}	t_{62}^1	10	t_{62}^2	30	t_{62}^3	10	50
P_{63}	t_{63}^1	30	t_{63}^2	20			50

algorithm. The state $s_0 = \langle \text{NULL}, \text{NULL}, \ldots, \text{NULL} \rangle$ is the initial state of the algorithm and the states $s_F = \langle P_{1j_1}, P_{2j_2}, \ldots, P_{nj_n} \rangle$ with $P_{ij_i} \neq \text{NULL}$, for all i, are the final states.

The A* algorithm starts from the initial state s_0 and finds a final state s_F such that the cost of getting from s_0 to s_F is minimal among all paths leading from s_0 to any final state. The cost of such a path is the total cost required for processing all n queries. Given a state s, we will denote by $scost(s)$ the cost of getting from the initial state s_0 to s.

In order for an A* algorithm to have fast convergence, a *lower bound* function h is introduced on states. This function is used to prune down the size of the search space that will be explored. If the algorithm of Grant and Minker [9] is modeled under the framework we just proposed, that is as an A* algorithm over the specific state space, the function $h: \mathcal{S} \rightarrow \mathbb{Z}$ applied on a given state $s = \langle P_{1k_1}, P_{2k_2}, \cdots, P_{nk_n} \rangle$ will be

$$h(s) = \left(\sum_{i=1}^{next(s)-1} est\_cost(P_{ik_i}) \right) + \left(\sum_{i=next(s)}^{n} \min_{j_i} [est\_cost(P_{ij_i})] \right) - scost(s) \quad (5)$$

The function *est_cost* is defined on tasks as follows

$$est\_cost(t) = \frac{cost(t)}{n_q} \quad (6)$$

where n_q is the number of queries the task t occurs in and *cost* is the cost function on tasks that was introduced in Section 3. The idea behind defining such a function is that the cost of a task is amortized among the various queries that will *probabily* make use of it. For a plan P_{ij_i}, it is assumed that

$$est\_cost(P_{ij_i}) = \sum_{t \in P_{ij_i}} est\_cost(t) \quad (7)$$

It is easy to see that $est\_cost(P_{ij_i}) \leq Cost(P_{ij_i})$, the cost of plan P_{ij_i} as defined in equation (1), and therefore the A* algorithm is guaranteed to converge to an optimal solution [21]. Let us give an example, also drawn from [9], which will motivate the dicussion that follows.

Example 5. Suppose two queries Q_5 and Q_6, are given along with their plans: P_{51}, P_{52}, P_{61}, P_{62}, P_{63}. We will use t_{ij}^k to indicate the kth task of plan P_{ij}. Table I gives the costs for the tasks involved in each plan, and the

identical tasks are

$$t_{51}^1 \equiv t_{61}^1; \quad t_{51}^2 \equiv t_{62}^2; \quad t_{52}^2 \equiv t_{63}^2;$$

Given the actual task costs and the sets of identical tasks, the estimated costs (*est_cost*) for these tasks are:

Table II. Estimated Cost for the Tasks

Task	t_{51}^1	t_{51}^2	t_{51}^3	t_{52}^1	t_{52}^2	t_{61}^2	t_{61}^3	t_{62}^1	t_{62}^3	t_{63}^1
Estimated cost	20	15	5	35	10	10	5	10	10	30

and the estimated costs for the plans are:

Table III. Estimated Cost for the Plans

Plan	P_{51}	P_{52}	P_{61}	P_{62}	P_{63}
Estimated cost	40	45	35	35	40

Based on the above numbers and the construction procedure outlined, Figure 7 shows the search space $\mathcal{S}$ along with the costs of transitions between states and estimated costs of going from intermediate to final states. Tracing the A* algorithm we get

$s_0 = \langle \text{NULL, NULL} \rangle$ /* *expand state* s_0 */
$s_1 = \langle P_{51}, \text{NULL} \rangle$ /* *expand state* s_1 */
$s_2 = \langle P_{52}, \text{NULL} \rangle$ /* *expand state* s_2 */
$s_F = \langle P_{52}, P_{63} \rangle$ /* *the final solution* */

yielding $\langle P_{52}, P_{63} \rangle$ as the best solution. Notice that with this set of estimators the algorithm exhaustively searches all possible paths in the state space.

It is exactly this bad behavior of the algorithm that we will try to improve by examining more closely the relationships among various tasks. For example, in the case presented above, it is clear right from the beginning that plan P_{51} will not be able to share both of its tasks t_{51}^1 and t_{51}^2 with plans P_{61} and P_{62}, respectively, since *only one* of these two latter plans will be in the final solution (final state). Therefore, the value $est\_cost(P_{51})$ is less than what could be predicted after looking more carefully at the query plans. It is a known theorem, in the case of A* algorithms, that with a higher estimator the algorithm will take (at most) as many steps as with a lower one (see [21], Result 6, p. 81). Hence, estimating the cost function better will enable the algorithm to converge faster to the final solution.

We have developed an algorithm that, given a set of queries, their plans, and the set of identical tasks, computes a "good" estimator function. Using a graph model, we identify which plans are impossible to coexist in the final state reached by the A* algorithm. Then, the lower bound function h is defined in a way that will assign high cost to such plans, hence making them unlikely to be con-

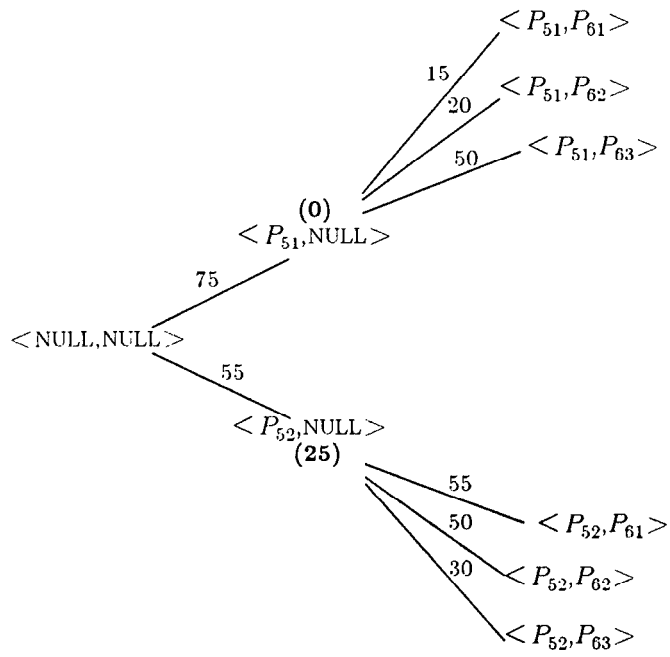

Fig. 7. Example search space for A* algorithm (numbers in parentheses show lower bound function values).

sidered during the search. Due to lack of space, this algorithm is presented in Appendix 1. We show here how the result of the preprocessing phase improves the performance of the algorithm.

Example 6. Suppose the two queries, Q_5 and Q_6, of Example 5 are given. The new estimators of plan costs will be derived based on the preprocessing algorithm. Given the costs as in Example 5, the algorithm of Appendix 1 computes the following (estimate) costs for the plans:

Table IV. Computation of (Estimated) Costs for the Plans from the Algorithm in Appendix 1

Plan	P_{51}	P_{52}	P_{61}	P_{62}	P_{63}
Estimated cost	55	45	35	35	40

Notice that the cost of plan P_{51} was underestimated by the Grant and Minker formula. Tracing the A* algorithm, we see that it explores the following states

$s_0 = \langle \text{NULL, NULL} \rangle$ /* *expand state* s_0 */
$s_1 = \langle P_{52}, \text{NULL} \rangle$ /* *expand state* s_1 */
$s_F = \langle P_{52}, P_{63} \rangle$ /* *the final solution* */

yielding again $\langle P_{52}, P_{63} \rangle$ as the optimal solution with cost 85. Notice that if the commands were executed sequentially it would have cost $Cost(P_{52}) + Cost(P_{63}) = 105$. Therefore, a total savings of 19% was achieved using the global optimization algorithm. Moreover, compared to the trace of the previous subsection, it can be seen that exhaustive search is avoided because of the high cost estimates for some paths.

We can summarize algorithm **HA** as follows: First, for all queries that do not share any task results with other queries, we find the originally cheapest plan and put it in the final processing sequence ε. For the rest of the queries, the **HA** algorithm is used to construct the global access plan:

HA1. ⟨*Estimate Plan Costs*⟩ Apply the preprocessing algorithm (described in Appendix 1) to obtain a good lower bound function h.

HA2. ⟨*Run A* Algorithm*⟩ Run the A* algorithm described above to obtain the execution plans.

HA3. ⟨*Find Global Access Plan*⟩ Let $\mathscr{P}$ be the set of all plans derived from the previous step. Integrate these plans to obtain the final global access plan.

The integrating process in step **HA3** is very similar to the one described for the interleaved execution algorithm where local plan graphs are merged together. Examining the estimated cost of the global access plan, we have

$$Cost(GP) = \sum_{P \in \mathscr{P}} Cost(P) - \sum_{s \in CS} savings(s) \tag{8}$$

where CS represents the *total* number of subexpressions found among the n plans in the final state s_F (not necessarily locally optimal) and $savings(s)$ is the cost savings function defined by eq. (4). Regarding the complexity of the algorithm **HA**, we must notice that it is very hard to analyze the behavior of an A* algorithm and give a very good estimate on the time required. In the worst case, of course, it may require time exponential on the number of queries, but on the average the complexity depends on how close the lower-bound function estimates the actual cost. However, the A* algorithm with the new estimator function we proposed will not take more steps than the originally suggested A* algorithm. This is based on the fact that for any plan P it is true that the estimator function $est\_cost(P)$, computed by the algorithm of Appendix 1, is greater than or equal to the one suggested in eq. (7). Given the definition of $h(s)$ in eq. (5), this means that the lower-bound function is also better. Therefore, as mentioned above, with the help of a known theorem from [21] our algorithm will give a solution in *at most* the same number of steps as the Grant and Minker algorithm.

Finally, note that the algorithm described is correct only in the cases where queries use solely equijoins and equality selection clauses. If arbitrary selection clauses are used, the A* algorithm presented above will not find the optimal solution. This is true because the imposed order in which the state vectors are filled (i.e., in ascending query index) may not result in the best utilization of common subexpression results. As an example, consider two queries, Q_1 and Q_2, such that Q_1 has a more restrictive selection than Q_2. Then clearly it would be better to consider executing Q_2 first since, in that case, the result of Q_2 can be

used to answer Q_1, the opposite being impossible. This problem with the heuristic algorithm can be easily fixed by changing the transitions to fill not the next available NULL slot in a state s, as it was done before through the use of $next(s)$ (see Definition 6), but rather any available (NULL) position of s. This results in larger fanout for each state and clearly more processing for the A* algorithm. The cost function *est_cost* is defined similarly with the difference that, in addition to identical tasks, pairs of tasks t_i and t_j such that $t_i \Rightarrow t_j$ and $t_j \not\Rightarrow t_i$ must be considered as well. However, the general algorithm we suggested can still be used; it is only the transitions between states and cost functions that need be adapted.

7. SOME EXPERIMENTAL RESULTS

We expect that for a large number of applications and query environments multiple-query optimization will offer substantial improvement to the performance of the system. In a series of experiments, we have simulated these algorithms using EQUEL/C [22] and the version of INGRES that is commercially available. The experiments were run over a slightly modified version of the set of queries that Finkelstein used in [7]. The reason such a set was chosen was primarily because Finkelstein's example was realistic and secondly because it can be used to expose all interesting parameters of the problem (see the discussion that follows). The database schema used was modeling a world of employees, corporations, and schools that the employees have attended, the relations being Employees, Corporations, and Schools, respectively. All eight queries, along with a brief description of the data they return, are shown in Appendix 2. Seven different sets of queries QSET1–QSET7 were formed by randomly choosing queries out of the original set, shown in Appendix 2. The queries within each of these sets were processed

(a) as independent queries;
(b) as the Interleaved Execution algorithm suggests; and, finally,
(c) as the Heuristic algorithm suggests.

Table V describes some characteristics of the sets QSET1 to QSET7. The second column indicates the number of queries used in each set, and the third column shows which queries from Appendix 2 were specifically used.

The above sets of queries were tested in various settings. First, unstructured relations were used with their sizes varied according to Table VI. Second, the same experiments were performed with structured relations. Specifically, the following structures were used

isam secondary index on Employees(experience)
isam primary structure on Corporations(earnings)
hash primary structure on Schools(sname)

The above choices were made in order to make locally optimal plans as cheap as possible. Finally, in another series of experiments the given queries were slightly modified by changing the constants used in one-variable selection clauses. The goal was to introduce higher sharing among the queries. Higher sharing is achieved when more queries can take advantage of the same temporary result.

Table V. Query Sets Used in Experiments

Query set	Number of queries	Queries
QSET1	2	{1, 7}
QSET2	2	{1, 6}
QSET3	4	{1, 2, 6, 7}
QSET4	2	{6, 7}
QSET5	4	{2, 3, 4, 6}
QSET6	7	{1, 2, 3, 4, 5, 6, 7}
QSET7	2	{7, 8}

Table VI. Sizes of Relations

Relation	Number of tuples
Employees	100-200-500-1000-10000
Corporations	10-20-50-100-500
Schools	20 (*fixed*)

Recall that the formula that provides an estimate on the cost savings using a global optimization algorithm is (for n queries $Q_1, \ldots, Q_n$)

$$\sum_{i=1}^{n} Bestcost(Q_i) - \sum_{s \in CS} savings(s)$$

where CS is the set of common temporary results. Therefore, higher cost reduction is achieved if more queries can use the same temporary result. By changing the constants in the qualification of the queries it was possible to check how the size of CS (i.e., the number of common subexpressions) affected the cost of processing the global access plans.

The measure used in this performance study was

$$PERCI = \frac{Cost_1(I/O) - Cost_2(I/O)}{Cost_1(I/O)} \cdot 100\% \tag{9}$$

where $Cost_1(I/O)$ is the number of I/Os required to process all queries assuming no global optimization is performed. This is the cost of locally optimal plans generated by the optimizer and assuming that temporary results are always built. $Cost_2(I/O)$ is the corresponding figure in the case where a global access plan is constructed according to some of the presented optimization algorithms. *PERCI* stands for *PERC*entage of *I*mprovement. The analogous CPU measure was also recorded; however, the numbers were almost the same and will not be shown. In the following, the results of the experiments are described in detail.

7.1 Unstructured Relations

Because of the similarity of the results we will group the diagrams according to the differences observed among the outcomes of the algorithms used for optimization. Two diagrams are presented: one for query sets QSET1–QSET6 and

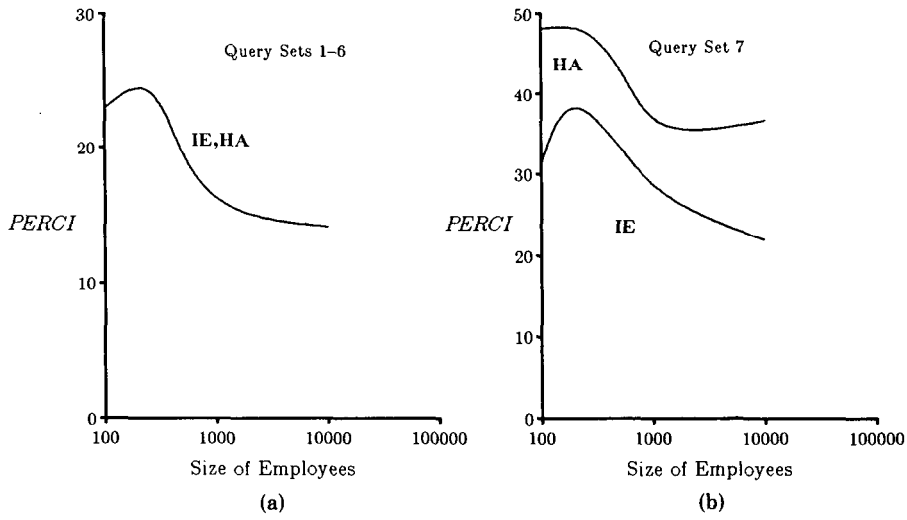

Fig. 8. Improvements for unstructured relations: Query sets 1–6 and 7.

another for QSET7. For query sets in the first group both **IE** and **HA** algorithms gave exactly the same results (in the sense that the global access plan was the same). The second group gave different results for the **IE** and **HA** algorithms. Figures 8(a) and 8(b) illustrate how *PERCI* varies for the two above-mentioned groups according to the size of the database in the case of unstructured relations. The size of the database is represented by the size of the Employees relation. The reasons for choosing that relation was first that all queries were using Employees (compared to Corporations or Schools) and second the fact that the diagrams are similar for the Corporations relation as well.

Some comments can be made here for these diagrams. First, there is always a gain in performance by doing multiple-query optimization, i.e., $PERCI \geq 0$, in all the experiments run, due to the overlap among the queries. Second, after some size of the relations, *PERCI* starts to decrease. This was due to the specific type of queries used. In particular, because of queries involving joins, the denominator of formula (9) grows faster than the numerator. In the given queries, the selection clauses were responsible for the savings in the numerator. That savings increases with rate proportional to the factor by which a relation is reduced as a result of performing a restriction on it (i.e., $1 - S$, where S is the selectivity of the selection clause). On the other hand, if joins are included in the queries, $Cost_1(I/O)$ increases with a rate that depends on the cost of the join operation. It turns out that for small sizes of the relations the latter factor is less than the former, while after some size this relationship is reversed. Hence, the slight increase followed by a decrease in the values of *PERCI* indicated in the above diagrams.

Finally, for the last query set QSET7, the plan generated by **HA** was significantly better than the one generated by **IE**. By allowing the result of the join e.employer = c.cname to be shared by both queries 7 and 8, significantly better performance was achieved.

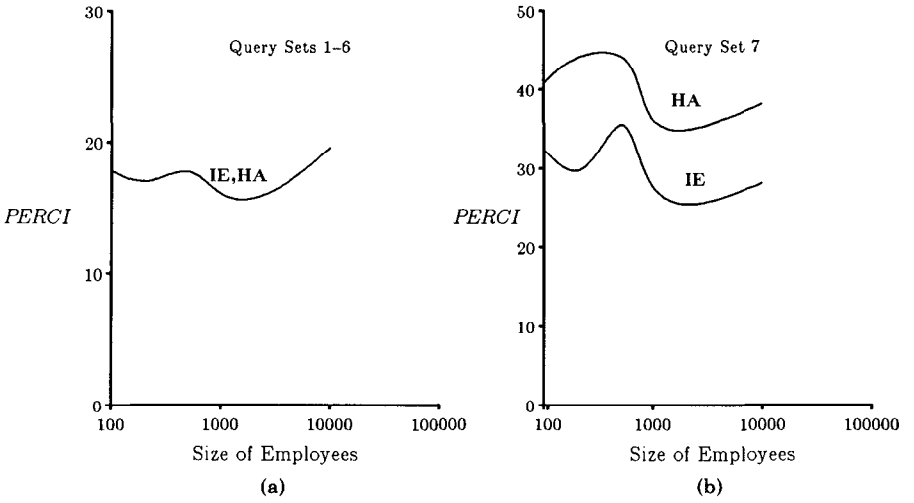

Fig. 9. Improvement for structured relations: Query sets 1–6 and 7.

7.2 Structured Relations

The same set of experiments was run over a structured database. Relations were indexed as mentioned in the beginning of this section. The reason for doing these experiments was to check if the overhead of accessing a relation through a secondary structure might be higher than the overhead of accessing an unstructured intermediate result. For example, suppose that retrieving the part of a relation that satisfies a simple one-variable restriction requires 10 page accesses. That includes the cost of searching first the index table and then accessing the data pages. Suppose now that there is an intermediate result, produced by some other query, that can be used to answer the same restriction clause. If the size of that intermediate result is less than 10 pages, then it will be more efficient to process the restriction by scanning the unstructured temporary result than going through the index table.

Figures 9(a) and 9(b) illustrate how *PERCI* varies for the two above-mentioned groups according to the size of the database in the case of structured relations. Comparing the values of *PERCI* with the corresponding ones of the previous subsection, we can observe some decrease of 10–20% for **IE** and **HA** depending on the size of the involved relations. This was expected since using indexes reduces $Cost_1(I/O)$. However, after some size of the Employee relation, *PERCI* starts increasing instead of decreasing, which was the case in the experiments of the previous subsection. This behavior is due to the fact we mentioned above (i.e., the overhead involved in using an index to access a relation). Moreover, the above effect is more obvious in cases where the involved relations are large. Then the size of the secondary indexes is in many cases significantly larger than the sizes of temporary results. Notice also that for small sizes of the Employee relation, *PERCI* is decreasing. That was expected because for small relations temporary results grow faster in size than the index tables. Finally, we notice that the relative performance of the three algorithms is not affected by the existence of indexes (i.e., **HA** still performs better than **IE**).

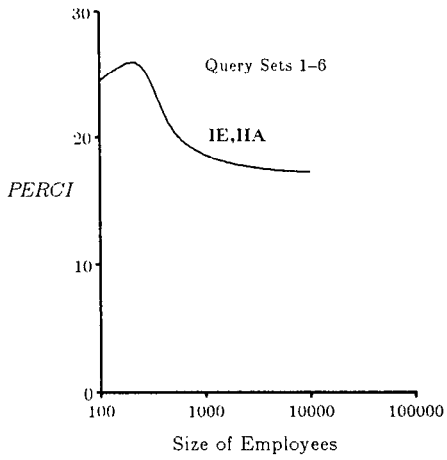

Fig. 10. Performance improvement for higher sharing.

7.3 Higher Sharing

In this last experiment, the given query sets were run over the same database with a modification in the queries so that higher degree of sharing is possible. That effect was introduced by changing the restrictions experience ≥ 20 found in queries 2, 4, 5, and 7 to experience ≥ 10. This way, the same temporary result could be used in the evaluation of more queries, compared to the ones in the experiments of the previous two subsections. Figure 10 illustrates how *PERCI* varied with the size of the database in the case of unstructured relations and for the first group of query sets (i.e., QSET1–QSET6). Query set 7 was not affected by this modification in the selection clauses in the sense that no increase in sharing was possible. Notice that the curve is similar to the one of Figure 8. However, because of the higher degree of sharing among queries, an increase of about 10% in the performance improvement was observed.

8. SUMMARY

The first major contribution of this paper lies in the presentation of a set of algorithms that can be used for multiple-query processing. Although some relevant work has been done in the past, we provide the first systematic way of designing multiple-query processing algorithms. The main motivation for performing interquery analysis is the fact that common intermediate results may be shared among various queries. We showed that various algorithms can be used for multiple-query optimization. More sophisticated algorithms (like **HA**) can be used to give better access plans at the expense of increased complexity of the algorithm itself.

Some of the algorithms proposed were based simply on the idea of reusing temporary results from the execution of queries, where the processing of each individual query is based on a locally optimal plan. Using plans instead of queries enabled us to concentrate on the problem of using efficiently common results rather than isolating common subexpressions. The heuristic search algorithm provides a general framework for the design of optimization algorithms. As an example, we have shown how the algorithm by Grant and Minker can be modeled

under this framework. In addition, we have suggested a preprocessing phase that derives a better cost estimator function to be used by the A* algorithm.

In general, the result of a global optimization algorithm should always be compared to what a conventional optimizer can do and the cheapest processing schedule should be processed. We expect that for a large number of applications and query environments multiple-query optimization will offer substantial improvement to the performance of the system. The experimental results described in Section 7 are the second major contribution of our work. They constitute the first empirical results in the area. In a series of experiments, we have simulated these algorithms and checked the performance of the resulting global access plans under various database sizes and physical designs. This enabled us to check the usefulness of these algorithms even in the presence of fast access paths for relations. The results were very encouraging and showed a decrease of 20–50% in both I/O and CPU time.

As interesting future research directions in the area of multiple-query optimization we view the development of efficient algorithms for common subexpression identification and the extension of the algorithms presented to cover more general predicates. In addition, we currently focus on developing an analytical model for a multiple-query processing environment. The experimental results of Section 7 agree with our preliminary analytical results but there is more work that need be done in this direction. Using a good analytical model will allow us to simulate various environments with different query mixes.

In a different direction, we view the application of our method in rule-based systems as a very interesting problem for investigation. For example, Prolog and database systems based on logic [29] can easily be extended to perform multiple-query optimization. Finally, some of the techniques that we developed here can be applied in processing recursion in database environments [14]. This is mainly due to the fact that in evaluating recursive queries one usually processes iteratively similar operations. These operations often access the same data, for the relations accessed are always the same. Investigating how our algorithms can be used in this recursive query processing environment seems to be a very interesting problem for future research.

APPENDIX 1

The goal of this appendix is to describe a preprocessing step that computes a better lower bound function for the A* algorithm of Section 6. Suppose that n sets of plans $\mathscr{P}_1, \mathscr{P}_2, \ldots, \mathscr{P}_n$ are given, with $\mathscr{P}_i = \{P_{i1}, P_{i2}, \ldots, P_{ik}\}$ (for simplicity, instead of P_{ik_i}, we use P_{ik} to denote plans). Let also t_{ij}^k denote the kth task of plan P_{ij}. We also assume that the pairs of identical tasks are given. We then define a directed graph $G(V, A)$ in the following way:

—For each plan P_{ij} that has a task t_{ij}^k identical to task(s) used for evaluating other than the ith query, introduce a vertex v_{ij}.

—For each pair $t_{ij}^r \in P_{ij}$, $t_{kl}^s \in P_{kl}$ of such identical tasks there is an arc connecting the two vertices $(v_{ij} \rightarrow v_{kl})$ if there is no other plan P_{km} with a task t_{km}^q such that for some u, $t_{km}^q \equiv t_{kl}^u$.

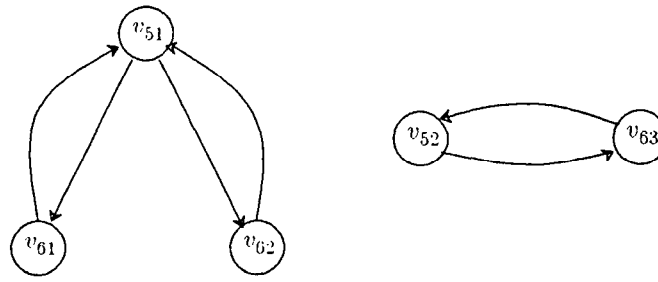

Fig. 11. Graph G for queries Q_5 and Q_6.

Given the above definition, a unique graph can be built based on a set of plans and a set of identities among tasks. Notice that *not all* plans are needed to build the graph. Only those having identical tasks among them are considered. Also, there may be more than one directed edge $(v_{ij} \rightarrow v_{kl})$ going from v_{ij} to v_{kl} if there are more than one pair of identical tasks involved in plans P_{ij} and P_{kl}. In order to reduce the size of the graph, only one edge $v_{ij} \rightarrow v_{kl}$ is recorded for any two vertices v_{ij} and v_{kl} that have at least one edge between them. No information is lost that way. The number of identical tasks found between the two plans is of no importance.

The goal of the preprocessing phase is to find plans that are most probably not sharing their tasks with other plans. The algorithm used is a slightly modified Depth-First-Search (DFS) algorithm. The difference is that in the course of backing up to the vertex v_{ij} from which another vertex v_{kl} was reached using the edge $v_{ij} \rightarrow v_{kl}$, the identification (subscript) kl is stored in some set associated with vertex v_{ij}. Call that set the *Need* set of vertex v_{ij}. Then, at the end of the algorithm, delete from G all vertices that have two or more members $k'l'$ and kl in their *Need* sets, such that $k' = k$. Along with the vertex, its edges (both out- and in-going) are also marked as OUT. This deletion process is continued by deleting vertices that have at least one out-going edge marked OUT. The edge and vertex elimination process stops when no more deletions are possible. Call the final graph $G'(V', A')$ and let $\mathscr{P}'$ be the set of plans P_{ij} that have a corresponding vertex v_{ij} in G'.

What is achieved through that preprocessing phase is the considerable reduction of the size of the search space explored by the A* algorithm. Only plans in $\mathscr{P}'$ are considered in order to derive the *est_cost* values. To give an example of the preprocessing phase, we apply the above procedure on Example 5 of Section 6.

Example 7. We are given again the same two queries Q_5 and Q_6 and five plans: P_{51}, P_{52}, P_{61}, P_{62}, P_{63}. The graph of Figure 11 gives the graph G for the set of plans given.

Suppose that the depth-first-search procedure starts from v_{51} and v_{52} for the left and right part of the graph of Figure 11, respectively. After the DFS has

Fig. 12. Final graph G'.

been performed, the *Need* sets for the various vertices will be as shown in Table VII below:

Table VII. *Need* Set After DFS Has Been Performed

Vertex	Need
v_{51}	{51, 61, 62}
v_{52}	{52, 63}
v_{61}	{51, 61}
v_{62}	{51, 62}
v_{63}	{52, 63}

From the above table it can be seen that vertex v_{51} must be eliminated since it can reach both v_{61} and v_{62} through directed paths. After that, the edges ($v_{51} \rightarrow v_{61}$), ($v_{61} \rightarrow v_{51}$), ($v_{51} \rightarrow v_{62}$), and ($v_{62} \rightarrow v_{51}$) are marked as OUT. This causes vertices v_{61} and v_{62} to be deleted also. No more vertices can be deleted. The remaining graph is shown in Figure 12.

Finally, $\mathscr{P}' = \{P_{52}, P_{63}\}$.

Using the result of the preprocessing phase, we next compute the new estimated costs for tasks and plans. First, based on the cost function *cost* defined for tasks, the following function *coalesced_cost* on tasks t [9] is defined (*coalesced_cost* is identical to the *est_cost* function of Section 6):

$$coalesced\_cost(t) = \frac{cost(t)}{n_q} \tag{10}$$

where n_q is the number of queries task t occurs in, and for plans

$$coalesced\_cost(P_{ij}) = \sum_{t \in P_{ij}} coalesced\_cost(t) \tag{11}$$

Now, given a plan P_{ij} and a specific task t_{ij}^k, let $\mathscr{Q}_{ij}$ be the set of queries Q_l, $l \neq i$, that have a plan that has a common task with P_{ij}. Also, let n_{ij}^l be the number of plans P_{lr} that correspond to query Q_l in $\mathscr{Q}_{ij}$. Then, *est_cost* is defined as follows

(a) If the plan P_{ij} is not in $\mathscr{P}'$ and $n_{ij}^l > 1$ for at least one query Q_l, then

$$est\_cost(P_{ij}) = Cost(P_{ij}) - \sum_{Q_l \in \mathscr{Q}_{ij}} \max[coalesced\_cost(t_{ij}^k)] \tag{12}$$

where $t_{ij}^k \equiv t_{lr}^s$, for some r and s.

(b) If the plan is in $\mathscr{P}'$ or it is not in $\mathscr{P}'$ but the above condition on n_{ij}^l does not hold, then

$$est\_cost(P_{ij}) = coalesced\_cost(P_{ij})$$

If we consider the queries of example 7, the above preprocessing algorithm provides the following estimated costs (see Section 6):

Table VIII. Estimated Cost

Plan	P_{51}	P_{52}	P_{61}	P_{62}	P_{63}
Estimated cost	55	45	35	35	40

Notice that the new values are greater than or equal to the ones derived by Grant and Minker, thus guaranteeing (a) on the average, less and (b) in the worst case, the same number of steps for the A* algorithm.

APPENDIX 2

The experiments described in Section 7 were run over the database

Employees (name, employer, age, experience, salary, education)
Corporations (cname, location, earnings, president, business)
Schools (sname, level)

The set of queries used, expressed in QUEL, is shown next. Assuming,

```
range of e is Employees
range of c is Corporations
range of s is Schools
```

Q1. *Get all employees with* 10 *years of experience or more*

```
retrieve (e.all) where e.experience ≥ 10
```

Q2. *Get all employees* 65 *years old or less with* 20 *years of experience or more*

```
retrieve (e.all) where e.experience ≥ 20 and e.age ≤ 65
```

Q3. *Get all pairs (employee, corporation), where the employee has* 10 *years of experience or more, and works in a corporation with earnings more than* 500*K and located anywhere but in Kansas.*

```
retrieve (e.all, c.all)
  where e.experience ≥ 10 and e.employer = c.cname
  and c.location ≠ "KANSAS" and c.earnings > 500
```

Q4. *Get all pairs (employee, corporation), where the employee has* 20 *years of experience or more, and works in a corporation with earnings more than* 300*K and located anywhere but in Kansas*

```
retrieve (e.all, c.all)
  where e.experience ≥ 20 and e.employer = c.cname
  and c.location ≠ "KANSAS" and c.earnings > 300
```

Q5. *Get all pairs (president, corporation), where the president is* 65 *years old or younger, with* 20 *years of experience or more, and the corporation is located in NEW YORK and has earnings more than* 500*K*

```
retrieve (e.all, c.all)
  where e.experience ≥ 20 and e.age ≤ 65
  and e.employer = c.cname and e.name = c.president
  and c.location = "NEW YORK" and c.earnings > 500
```

Q6. *Get all pairs (president, corporation), where the president is 60 years old or younger, with 30 years of experience or more, and the corporation is located in NEW YORK and has earnings more than 300K*

```
retrieve (e.all, c.all)
  where e.experience ≥ 30 and e.age ≤ 60
  and e.employer = c.cname and e.name = c.president
  and c.location = "NEW YORK" and c.earnings > 300
```

Q7. *Get all triples (employee, corporation, school) where the employee is 65 years old or younger, has 20 years of experience or more and holds a university degree working for a corporation located in NEW YORK and with earnings more than 500K*

```
retrieve (e.all, c.all, s.all)
  where e.experience ≥ 20 and e.age ≤ 65
  and e.employer = c.cname
  and c.location = "NEW YORK" and c.earnings > 500
  and e.education = s.sname and s.level = "univ"
```

Q8. *Get all pairs (employee, corporation), where the employee is 65 years old or younger, with 20 years of experience or more and the corporation is located in NEW YORK and has earnings more than 300K*

```
retrieve (e.all, c.all)
  where e.experience ≥ 20 and e.age ≤ 65
  and e.employer = c.cname
  and c.location = "NEW YORK" and c.earnings > 300
```

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Michael Stonebraker, for giving me the opportunity to work in the area of multiple-query processing and for providing many helpful comments on an earlier draft. My colleague, Yannis Ioannidis, and the anonymous referees have provided criticisms and suggestions that have greatly improved the presentation of this paper. Finally, I would like to acknowledge the Systems Research Center of the University of Maryland for its partial support through the National Science Foundation grant CDR-85-00108.

REFERENCES

1. ASTRAHAN, M. ET AL. System R: A relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
2. CHAKRAVARTHY, U. S., FISHMAN, D. H., AND MINKER, J. Semantic query optimization in expert systems and database systems. In *Expert Database Systems: Proceedings From the 1st International Workshop*, L. Kershberg, Ed. Benjamin/Cummings, Menlo Park, Calif. 1986, 659–674.
3. CHAKRAVARTHY, U. S., AND MINKER, J. Processing multiple queries in database systems. *Database Eng. 5*, 3 (Sept. 1982), 38–44.
4. CHAKRAVARTHY, U. S., AND MINKER, J. Multiple query processing in deductive databases. Tech. Rep. TR-1554, Dept. of Computer Science, Univ. of Maryland, College Park, Md., Aug. 1985.
5. CHAKRAVARTHY, U. S., MINKER, J., AND GRANT, J. Semantic query optimization: additional constraints and control strategies. In *Proceedings of the 1st International Conference on Expert Database Systems* (Charleston, S. C., April 1986). Institute of Information Management and Policy, Univ. of South Carolina, Apr. 1986, 259–270.

6. CLOCKSIN, W., AND MELLISH, C. *Programming in PROLOG.* Springer-Verlag, New York, 1981.
7. FINKELSTEIN, S. Common expression analysis in database applications. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (Orlando, Fla., June 1982) ACM, New York, 1982, 235–245.
8. GALLAIRE, H., AND MINKER, J. *Logic and Data Bases.* Plenum Press, New York, 1978.
9. GRANT, J., AND MINKER, J. On optimizing the evaluation of a set of expressions. Tech. Rep. TR-916, Univ. of Maryland, College Park, Md., July 1980.
10. GRANT, J., AND MINKER, J. Optimization in deductive and conventional relational database systems. In *Advances in Data Base Theory, Vol. 1,* H. Gallaire, J. Minker, and J.-M. Nicolas, Eds. Plenum Press, New York, 1981, 195–234.
11. GUTTMAN, A. New features for relational database systems to support CAD applications. Ph.D. dissertation, Computer Science Div., Univ. of California, Berkeley, June 1984.
12. HALL, P. V. Common subexpression identification in general algebraic systems, Tech. Rep. UKSC 0060, IBM United Kingdom Scientific Centre, Nov. 1974.
13. HALL, P. V. Optimization of a single relational expression in a relational data base system. *IBM J. Res. Dev. 20,* 3 (May 1976), 244–257.
14. IOANNIDIS, Y. Processing recursion in deductive database systems. Ph.D. dissertation, Univ. of California, Berkeley, July 1986.
15. JARKE, M., CLIFFORD, J., AND VASSILIOU, Y. An optimizing PROLOG front-end to a relational query system. In *Proceedings of ACM-SIGMOD International Conference on the Management of Data* (Boston, Mass., June 18–21, 1984). ACM, New York, 1984, 296–306.
16. JARKE, M. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems,* W. Kim, D. Reiner, and D. Batory, Eds. Springer-Verlag, New York, 1984, 191–205.
17. KIM, W. Global optimization of relational queries: a first step. In *Query Processing in Database Systems,* W. Kim, D. Reiner, and D. Batory, Eds. Springer-Verlag, New York, 1984, 206–216.
18. KUNG, R., HANSON, E., IOANNIDIS, Y., SELLIS, T., SHAPIRO, L., AND STONEBRAKER, M. Heuristic search in data base systems. In *Expert Database Systems: Proceedings From the 1st International Workshop,* L. Kershberg, Ed. Benjamin/Cummings, Menlo Park, Calif., 1986, 537–548.
19. LARSON, P., AND YANG, H. Computing queries from derived relations. In *Proceedings of International Conference on Very Large Data Bases* (Stockholm, Aug. 1985), 259–269.
20. NAQVI, S., AND HENSCHEN, L. On compiling queries in recursive first-order databases. *J. ACM 31,* 1 (Jan. 1984), 47–85.
21. NILSSON, N. J. *Principles of Artificial Intelligence.* Tioga, Palo Alto, Calif., 1980.
22. RELATIONAL TECHNOLOGY, INC. *EQUEL/C User's Guide.* Version 2.1, Relational Technology, Inc., Berkeley, Calif., July 1984.
23. ROSENKRANTZ, D. J., AND HUNT, H. B. Processing conjunctive predicates and queries. In *Proceedings of the International Conference on Very Large Data Bases* (Montreal, Oct. 1980), 64–72.
24. ROUSSOPOULOS, N. View indexing in relational databases. *ACM Trans. Database Syst. 7,* 2 (June 1982), 258–290.
25. ROUSSOPOULOS, N. The logical access path schema of a database. *IEEE Trans. Softw. Eng. SE-8,* 6 (Nov. 1982), 563–573.
26. SELLIS, T., AND SHAPIRO, L. Optimization of extended database languages. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (Austin, Tex., May 1985), ACM, New York, 1985, 424–436.
27. STONEBRAKER, M., AND ROWE, L. The design of POSTGRES. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (Washington D. C., May 28–30, 1986). ACM, New York, 1986, 340–355.
28. STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. The design and implementation of INGRES. *ACM Trans. Database Syst. 1,* 3 (Sept. 1976), 189–222.
29. ULLMAN, J. Implementation of logical query languages for data bases. *ACM Trans. Database Syst. 10,* 3 (Sept. 1985), 289–321.

30. WONG, E., AND YOUSSEFI, K. Decomposition: A strategy for query processing. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 223–241.
31. ZANIOLO, C. The database language GEM. In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (San Jose, Calif., May, 1983). ACM, New York, 1983, 207–218.

Received September 1986; revised January 1987; accepted June 1987