GLOBAL QUERY OPTIMIZATION

by

Timos K. Selis

Memorandum No. UCB/ERL M86/19

3 March 1986

COVER PAGE

GLOBAL QUERY OPTIMIZATION

by

Timos K. Selis

Memorandum No. UCB/ERL M86/19

3 March 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

GLOBAL QUERY OPTIMIZATION

by

Timos K. Selis

Memorandum No. UCB/ERL M86/19

3 March 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

GLOBAL QUERY OPTIMIZATION

by

Timos K. Sellis

Department of Electrical Engineering and Computer Science
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

In some recently proposed extensions to relational database systems as well as in deductive databases, a database system is presented with a collection of queries to process instead of just one. It is an interesting problem then, to come up with algorithms that process these queries together instead of one query at a time. We examine the problem of multiple (global) query optimization in this paper. A hierarchy of algorithms that can be used for global query optimization is exhibited and analyzed. These algorithms range from an arbitrary serial execution without any sharing of common results among the queries to an exhaustive search of all possible ways to process all queries. Preliminary experimental results are also given.

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.

1. INTRODUCTION

To extend the benefits of the database approach to other than business data processing areas, like artificial intelligence and engineering design automation, many researchers have defined various extensions to existing database languages. Examples of these extended languages include the language QUEL* [KUNG84], designed to support artificial intelligence applications, GEM [ZANI83], to support a semantic data model, and the proposal of [GUTT84], for support of VLSI design. A significant part of extended database languages is support for multiple command processing. In [SELL85] we have 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).

There are many applications where more than one query are presented to the system in order to be processed. First, consider a database system enhanced with inference capabilities (*deductive database system)* [GALL78]. A single query given to such a system may result to more than one actual queries that will have to be run over the database. As an example, consider the following relation for employees

```
EMP (name,salary,experience,manager,dept_name)
```

Assume also the existence of a set of rules that define when an employee is well paid. We will express these rules in terms of `retrieve` commands.

```
   /* An employee is well paid if he/she makes more than 40K */
Rule 1:   retrieve (EMP.all)  where EMP.salary > 40
   /* An employee is well paid if he/she makes more than 35K
      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 30K
      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 a query that asks

Is Mike well paid?

will have to evaluate all three rules in order to come up with the answer. Because of the similarities that PROLOG [CLOC81] 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 to be processed all together (we will see in the following what "all together" means). Finally, some proposals on processing recursion in database systems [NAQV84,IOAN86], suggest that a recursive Horn clause should be transformed to a set of other simpler Horn clauses (recursive and non-recursive). Therefore, the problem of multiple query processing arises in that environment as well. However, it is more complicated because of the presence of recursive queries.

Current query processors cannot optimize the execution of more than one queries. If given a set of queries, the common practice is to process each query separately. There are generally many possible ways of executing a query (*access plans*). For example, there may be a choice of indexes to use, or a choice of strategies for executing a relational operator such as the join. Access plans are simply sequences of such simple tasks as relation scans, index scans, etc. The query processor chooses the cheapest among these plans and then executes it to produce the result of the query. However, in the case where more than one queries are given at the same time 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 the tuples of 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.

This report discusses in more detail the ideas of [SELL86] and is organized as follows. Section 2 presents an overview of previous work done in similar problems while Section 3 first defines the query model that will be used throughout this paper and then presents a formulation for the multiple (or *global*) 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 global query optimization problem. Then, Sections 5 through 7 present these algorithms in more detail. Section 5 suggests an algorithm which finds a serial sequence for executing the queries with better performance than any other serial execution which executes the queries in an arbitrary order. Section 6 describes an algorithm that goes one step further by allowing the executions of the queries to interleave, while Section 7 proposes a more general heuristic algorithm. Finally, in Section 8 we present some experimental results and the last section concludes the presentation of the global query processing problem by summarizing our results and suggesting some areas for future research.

2. Previous Work

Problems similar to the problem of multiple query processing have been examined in the past in various contexts. Hall [HALL74,HALL76] 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 [GRAN80] and [GRAN81] 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 (i.e. at the time the queries are given to the system) 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 separately as groups instead of one at a time. During that stage common tasks are identified and sharing of the results of such tasks is used to reduce processing time.

Roussopoulos in [ROUS82a] and [ROUS82b] provides a framework for interquery analysis based on query graphs [WONG76], 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 ground relations. Indexes are then built as data structures to support fast processing of views.

Other researchers have also recently examined the problem of global query optimization. Chakravarthy and Minker [CHAK82,CHAK85] 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 [WONG76]. Using integrated query graphs, Chakravarthy and Minker suggest a generalization of the query decomposition algorithm of [WONG76]; however, this algorithm does not guarantee that the access plan constructed is the cheapest one possible. Kim in [KIM84] suggests also a two stage optimization procedure similar to the one in [GRAN81]. The unit of sharing among queries in Kim's proposal is the relation which is not always the best thing to assume, except in cases of single relation queries.

The work of [FINK82] and [LARS85] on the problem of deriving query results based on the results of other previously executed queries, is also related to the problem of global query optimization. The solutions suggested are useful to our analysis because they include efficient algorithms to detect common subexpressions among queries. These subexpressions characterize the data that is shared and accessed by more than one query. Jarke also discusses in [JARK84b] the problem of common subexpression isolation. He presents several different formulations of the same problem under various query language frameworks such as relational algebra, tuple calculus and relational calculus. In the same article he also describes how common expressions can be detected and used according to their type (e.g. single relation restrictions, joins, etc).

The main objective of our approach to multiple query processing is to use existing query optimizers as much as possible. We would like to avoid making significant changes to the query optimizer; instead, our goal is to provide a preprocessor that will reduce the execution cost as much as possible. This preprocessing phase is introduced as an extra step between the optimizer and the execution modules. However, since not all relational database systems have been designed based on 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 these two approaches. Architecture 1 can be used with minimal changes to existing

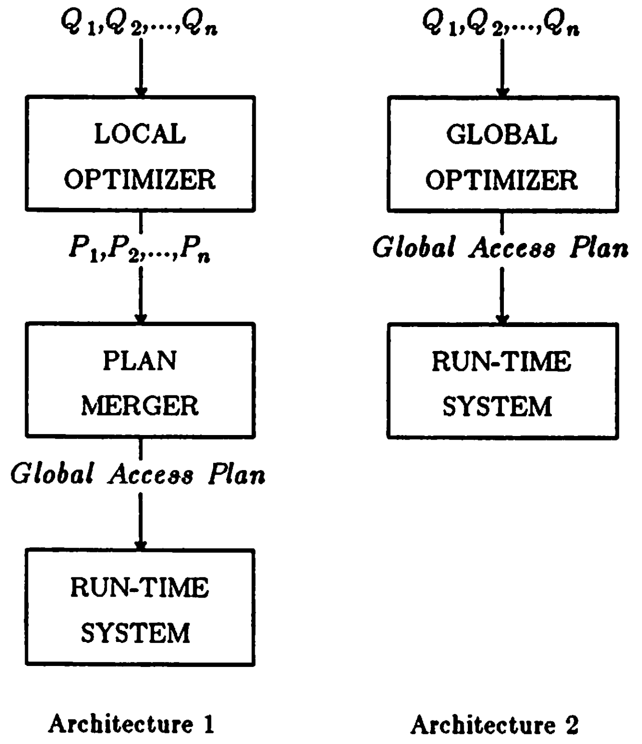


Figure 1: Multiple Query Processing Systems Architecture

optimizers. A conventional *Local Optimizer* generates one (*locally*) optimal access plan per query. The *Plan Merger* is a component which examines all *n* access plans and generates a larger plan, the global access plan, which is in turn processed by the *Run-Time System*. In many existing systems queries are *compiled* and saved in the form of access plans (see for example System-R [ASTR76] and POSTGRES [STON86]). It is then an interesting problem to derive procedures that, given a set of such plans, identify a sequence in which they must be run in order to reduce the I/O and/or CPU cost. More sophisticated procedures can also be used for that reason. For example, Chakravarthy and Minker [CHAK85] describe an algorithm to process multiple joins involving the same relation R by scanning R once and examining several restriction conditions in parallel. Using such a procedure though implies rewritting the query processor which, as we argued above, requires a major effort.

On the other hand, there are systems that do not store access plans for future reusal (e.g. INGRES [STON76]). 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. Architecture 2 therefore is not restricted to using locally optimal plans already stored in the system.

The purpose of the following sections is to exhibit a set of optimization algorithms that can be used for multiple query optimization either as Plan Mergers or as Global Optimizers. The algorithms to be presented differ on the complexity of the Plan Merger and on whether Architecture 1 or 2 is used. The trade offs 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* (or fields). A set of queries $\mathbf{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ on **D** is also given. 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 a field 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, A and B are fields of R_1 and R_2 respectively. 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 fields). 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 in its complexity. The restriction on conjunctive queries only is not a severe limitation since the result of a disjunctive query can be considered of as the union of the results of the disjuncts, i.e. each disjunct can be thought as a different query. Equijoins are also the only type of joins allowed among relations. This assumption is made in all the proposals mentioned in the previous section and seems quite natural considering the most common types of queries. Finally, not allowing projections enables us to concentrate on the problem of using effectively the results of common subexpressions rather than the problem of detecting if the result of a query can be used to compute the result of another query. Assuming projection lists, does not increase the complexity of the algorithms that perform multiple query optimization. It only increases the complexity of the algorithms that detect common subexpressions among queries. The proposals of [LARS85] and [FINK82] provide such algorithms.

A *task* is an expression *relname* $\leftarrow$ *expr*. *relname* is a 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 a conjunction of either selection predicates over the same relation or joins between two, possibly restricted, relations. This latter type covers queries that are processed not by performing the selections first followed by the join, but in a "pipelining" way. For example, consider the following query on the relations EMP (name,age,dept_name) and DEPT (dept_name,num_of_emps)

```
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 the query is by scanning the relation EMP and having each employee tuple with qualifying age be checked across the DEPT relation. There is no need in storing intermediate results for both EMP and DEPT. To be able to include this kind of processing in our model, the second type of join tasks was introduced. In the remaining discussion, tasks will be referred to as if they were simply the *expr* part, unless otherwise explicitly stated.

Let us define now a partial order on tasks. A task t_i *implies* task t_j ($t_i \Longrightarrow t_j$) iff t_i is a conjunction of selection predicates on attributes A_1, A_2, ..., A_k of some relation R, t_j is a conjunction of selection predicates on the same relation R and on attributes A_1, A_2, ..., 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.

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

a) *Selections* : $t_i \Longrightarrow t_j$ and $t_j \Longrightarrow t_i$

b) *Joins* : t_i is a conjunction of join predicates $E_1.A_1 = E_2.B_1$, $E_1.A_2 = E_2.B_2, \ldots, E_1.A_k = E_2.B_k$ and t_j is a conjunction of join predicates $E'_1.A_1 = E'_2.B_1$, $E'_1.A_2 = E'_2.B_2, \ldots, E'_1.A_k = E'_2.B_k$ where each of E_1, E_2, E'_1 and E'_2 is a conjunction of selections on a single relation and $E_1 \equiv E'_1$ and $E_2 \equiv E'_2$

Based on the above definition for tasks we now define the notion of an access plan. An *access plan* for a query Q is a sequence of tasks that produces the result of answering Q. Formally, an access plan is an acyclic directed graph $P=(V,E,L)$ (V, E and L being the sets of vertices, edges and vertex labels respectively) defined as follows :

- For every task t of the plan introduce a vertex v
- If the result of a task t_i is used in task t_j, introduce an edge $v_i \rightarrow v_j$ 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* $\leftarrow$ *expr*)

For example, consider the following query on the relations EMP (name,age,dept_name) and DEPT (dept_name,num_of_emps)

```
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 there are generally many possible plans that can be used in order to process a query.

Next we define a cost function *cost* : $V \rightarrow \mathbb{Z}$ for tasks. 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 have only I/O costs. Including CPU costs would only

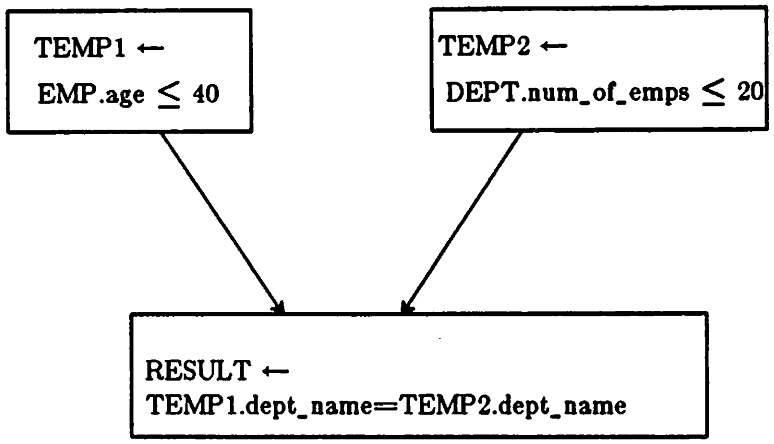


Figure 2: Example of an Access Plan

make the formulas more complex. Therefore,

$$cost(v_i) = \text{ the number of page accesses needed to process task } t_i$$

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

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

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. The union of the locally optimal plans is generally different than the globally optimal plan. Finally, for a given query Q, $Bestcost(Q)$ gives the cost of the (locally) optimal plan P. Hence, $Bestcost(Q) = \min_{p \in \mathbf{P}}[Cost(p)]$, where **P** is the set of all possible plans that can be used to evaluate Q.

Let us now consider a system that given a set **Q** of queries it 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 global query optimization becomes

> **Given** *n sets of access plans* $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n$, *with* $\mathbf{S}_i = \{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$ *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* $\mathbf{S}_i$*) 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. A Hierarchy of Algorithms

The primary source of redundancy in multiple query processing is 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 retrieving data from the database. Our intention here is to detect common subexpressions looking only at the logical expressions used in the descriptions of queries, that is by simply isolating pairs of expressions e_1 and e_2 where $e_1 \Longrightarrow e_2$. Therefore, detection of sharing is done at a high level using only the query expressions (qualifications) and without going to the actual data stored in the database. For example, e_1 may be `EMP.age` $\leq$ 30 and e_2 may be `EMP.age` $\leq$ 40. Then $e_1 \Longrightarrow e_2$. 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 are 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 $e_1 \Longrightarrow e_2$ without looking at the actual data stored [JARK84a,CHAK84,CHAK86]. Hence, query expressions are considered to be the only source for detecting common subexpressions. Because several algorithms have been published in the past on the problem of common subexpression isolation [ROSE80,FINK82,LARS85] we will not attempt here to present a similar algorithm. It is assumed that a procedure which decides, given two expressions e_1 and e_2, if $e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$, is available.

Second, as it was stated in the previous section, many systems store in the database optimal local access plans that have been produced in the past (e.g. System-R [ASTR76] and POSTGRES [STON86] choose to do so). Because the system cannot afford to store more than one plan for each query, it stores only locally optimal access plans. Then, if a set of queries is given, there is no need to generate new plans for those queries that have precomputed plans already stored in the database. However, for the rest of the queries, optimal plans are produced and saved for future reusal. When both precomputed and newly generated plans are available the global access plan is derived.

The various algorithms that can be used for global query optimization are grouped in a hierarchy shown in Figure 3. The reason the algorithms are organized in such a hierarchy is to

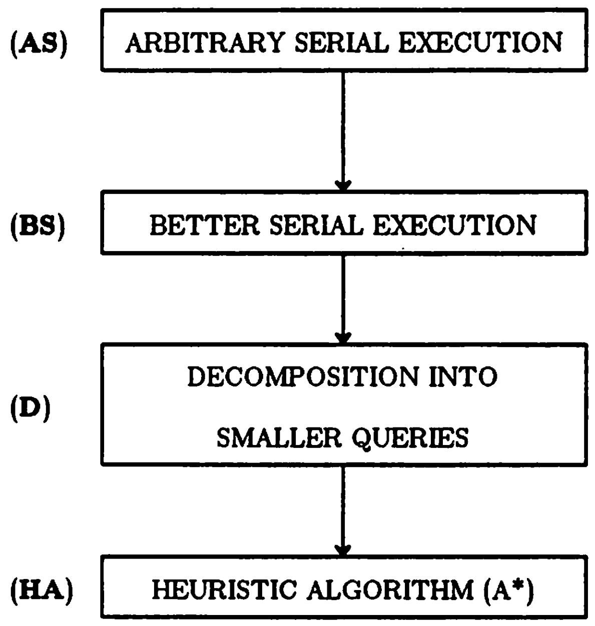


Figure 3: A Hierarchy of Multiple Query Processing Algorithms

indicate the interesting trade off between the time spent for optimization and the cost of executing the resulting global access plan. As we descend the hierarchy, the complexity of the algorithm increases while the access plan cost decreases. Algorithms **AS**, **BS** and **D** consider only access plans that are *locally* optimal. As mentioned above, the locally optimal plan for executing a query Q is derived by considering Q alone. Algorithm **AS** (Arbitrary Serial Execution) simply executes these plans in an arbitrary order. This corresponds to Architecture 1 of Figure 1 with the Plan Merger absent, i.e. no optimization is performed. Algorithm **BS** (Better Serial Execution) preprocesses the plans and generates a better order of execution so that intermediate results (temporaries) are reusable. In this case the Plan Merger of Figure 1 simply rearranges the order in which the plans are processed. Notice that in both algorithms **AS** and **BS** the unit of execution is a whole query, i.e. the second query is processed after the first one has been totally processed.

Algorithm **D** (Decomposition) presents a different paradigm. A query is decomposed into smaller subqueries which 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 result. As an example why **D** might be a better algorithm than **BS**, 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. If we run either Q_1 or Q_2 first we will be unable to use the intermediate results from the restrictions on EMP and DEPT effectively. However, the following global access plan is more efficient

```
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. It is drastically more efficient in the cases where restrictions reduce the sizes of the original relations significantly. The function of the Plan Merger, in the case of algorithm **D**, is to "glue" the plans together in a way that provides better utilization of common temporary (intermediate) results.

Finally, algorithm **HA** (Heuristic Algorithm) 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. Suppose we have 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
```

with optimal local plans

(P_3)
```
retrieve into TEMP1 (JOB.all,EMP.all)
    where JOB.job = EMP.job
    retrieve (TEMP1.all,DEPT.all)
    where TEMP1.dept_name = DEPT.dept_name
```

(P_4)
```
retrieve (EMP.all,DEPT.all)
    where EMP.dept_name = DEPT.dept_name
```

respectively. Notice that P_3 and P_4 do not share the common subexpression `EMP.dept_name=DEPT.dept_name`. Algorithm **HA** considers in addition to P_3 the plan that processes the join `EMP.dept_name=DEPT.dept_name`. It also uses some heuristics to reduce the number of permutations of plans it has to examine in order to find the optimal global plan. All the above algorithms are examined in more detail in the following three sections.

5. Serial Execution

Algorithms **AS** and **BS** of Figure 3 are based on some serial execution of the given queries $Q_1, Q_2, \ldots, Q_n$. As stated in the previous section we only consider the locally optimal plans P_i, $1 \leq i \leq n$. In the first case no restrictions are imposed on the order in which the queries are processed; that is what a conventional query processor would do. In the second case though some simple preprocessing is done aiming to better performance.

5.1.1. Arbitrary Serial Execution

In Algorithm **AS** the sequence in which the queries are run is chosen arbitrarily. We assume that all queries are processed without taking advantage of any common tasks that they may share. The global plan GP that is produced is simply the concatenation of the locally optimal plans for the queries in an arbitrary way. Therefore, for any order of processing $\mathbf{S} = \{Q_{i_1} Q_{i_2} \cdots Q_{i_n}\}$, with $Q_{i_k} \in \mathbf{Q}$ and all i_k distinct, the cost of the global access plan will be

$$Cost(GP) = \sum_{j=1}^{n} Bestcost(Q_{i_j})$$

As an example, consider the following queries Q_5 and Q_6

(Q_5)
```
retrieve (EMP.all,DEPT.all)
    where EMP.age ≤ 40
    and EMP.salary ≤ 10
    and EMP.dept_name = DEPT.dept_name
```

(Q_6)
```
retrieve (EMP.all,DEPT.all)
    where EMP.age ≤ 40
    and EMP.dept_name = DEPT.dept_name
```

Assume also that the sizes of the initial relations and temporary results are as follows

```
    size (EMP) = 100 pages
    size (DEPT) = 10 pages
    size (EMP.age≤40) = 20 pages
    size (EMP[age≤40 and salary ≤ 10]) = 10 pages
```

It is also assumed that the local plans for Q_5 and Q_6 store temporaries for the above restrictions. Then, processing S would require 110+C_j(10,10) page accesses for Q_5 and 120+C_j(20,10) page accesses for Q_6, where $C_j(a,b)$ is the cost of processing a join between two relations of sizes a and b pages. Hence, the total cost would be 230+C_j(10,10)+C_j(20,10) page accesses.

The above algorithm does not consider at all of reusing results that are produced as intermediate (temporary) relations. A simple extension would be to keep temporary relations after they are used so that subsequent queries may use them. Better than that, with some simple preprocessing we can find a serial execution that makes use of such temporary results. The next subsection presents such an approach.

5.1.2. Better Serial Execution

The goal of algorithm **BS** is to look at the optimal local plans and derive a serial execution schedule S that makes use of common subexpressions. Checking if a given temporary result can be used by another query is done through the procedure proposed in [FINK82].

The first step in deriving the execution schedule S builds a directed graph that will eventually suggest S using the directed paths of the graph. This kind of graph is very similar to the precedence graphs used in concurrency control [ULLM82] and it is used to indicate how the read set of one query is related to the read sets of other queries. First, queries that possibly have common subexpressions are identified. If some query Q_i does not share any of its input relations with any other query, it is put first in the sequence S. These queries are not amenable to any optimization other than what the locally optimal plan suggests. For the rest of the queries we define the following directed labeled graph $QG(V,E,L)$, with V being the set of vertices, E the set of edges and L a set of labels associated with edges

- For each plan $P_i(V_i,E_i,L_i)$ a node q_i is defined
- A directed edge $q_i \rightarrow q_j$ is introduced if

    a) *Proper Implication* : There are $v_i \in V_i$ and $v_j \in V_j$ such that $L_j(v_j) \Longrightarrow L_i(v_i)$ and $L_i(v_i) \not\Longrightarrow L_j(v_j)$

b) *Identical Nodes* : There are $v_i \in V_i$ and $v_j \in V_j$ such that $L_j(v_j) \equiv L_i(v_i)$ and $i < j$

- Assume that edge $q_i \rightarrow q_j$ is introduced because of nodes v_i of P_i and v_j of P_j respectively. Then the label of the edge $q_i \rightarrow q_j$ is the savings in the cost of executing $L_j(v_j)$ given the result of $L_i(v_i)$. This cost is estimated assuming that one or more of the relations used in v_j are substituted by the temporary relation that is created in the task v_i.

Edges of type (a) are introduced to indicate which queries (tail of an edge) can be used in the evaluation of other queries (head of an edge). The second rule for edge definition is introduced to break ties between identical expressions in a specified manner. Algorithm **BS** then proceeds in the following way :

[1] If multiple edges with the same direction are found between two nodes q_i and q_j replace them with a single edge with label the sum of the labels of the previous edges.

[2] If the resulting graph is acyclic then the execution order S is derived from the directed paths that are imposed on the graph.

[3] If the resulting graph has cycles, these are broken by omitting a set of edges with minimal sum of labels. S is then produced as in [2].

Let $QG'(V,E',L)$ be the resulting graph. The last step of the above algorithm is a well known NP-complete problem, known as "the feedback arc set problem" [GARE79]. However, in multiple query optimization the graph will have few nodes equal to the number of queries that access common data and not many cycles. Therefore, this problem has only minor effect on the performance of the algorithm. A simple analysis shows that the formula for computing the estimated cost of the global plan imposed by the sequence S is

$$\begin{aligned} Cost(GP) &= \sum_{i=1}^{n} Bestcost(Q_i) - \sum_{e \in E'} L(e) \\ &= \sum_{i=1}^{n} Bestcost(Q_i) - \sum_{s \in CS} n_s \cdot savings(s) \end{aligned}$$

where CS is the set of common subexpressions s found among the queries and used in the final graph QG', n_s is the number of times the result of a common subexpression s is used in the final sequence and $savings(s)$ is the cost that is saved if temporary results instead of ground relations are used. That cost is defined as follows:

Let R be a relation and s_1 and s_2 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. Then

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

In order to construct graph QG, the above algorithm requires time in the order of $\prod_{i=1}^{k} |V_i|$, where $k = |V|$ is the number of vertices of graph QG and V_i are the sets of vertices for plans P_i, $1 \le i \le k$. Step [3] is the most expensive step and in the worst case requires time exponential on the number of the edges.

Let us show with an example how **BS** works. Suppose that the queries Q_5 and Q_6 of the previous subsection are given. The directed graph constructed is shown in Figure 4.

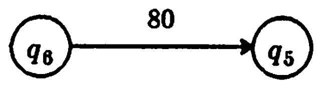


Figure 4: *QG* Graph for Queries Q_5 and Q_6

The edge $q_6 \rightarrow q_5$ is introduced because `[EMP.age ≤ 40 and EMP.salary ≤ 10]` $\Longrightarrow$ `EMP.age ≤ 40`. Therefore the serial execution will be S=$\{Q_6\ Q_5\}$ which uses 80 page accesses less than an arbitrary serial execution which was seen in the previous section, for a savings of 35%.

To give an example where a cyclic graph QG may occur, consider queries Q_1 and Q_2 of section 3

```
(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
```

with optimal local plans

```
(P1)    retrieve into tempEMP1(EMP.all)
            where EMP.age ≤ 40
        retrieve into tempDEPT1(DEPT.all)
```

```
          where DEPT.num_of_emps ≤ 20
      retrieve (tempEMP1.all,tempDEPT1.all)
          where tempEMP1.dept_name = tempDEPT1.dept_name

(P2)  retrieve into tempEMP2(EMP.all)
          where EMP.age ≤ 50
      retrieve into tempDEPT2(DEPT.all)
          where DEPT.num_of_emps ≤ 10
      retrieve (tempEMP2.all,tempDEPT2.all)
          where tempEMP2.dept_name = tempDEPT2.dept_name
```

and sizes of relations and intermediate results

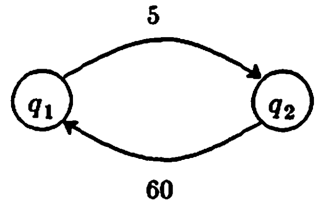
```
size (EMP) = 100 pages , size (DEPT) = 10 pages
size (tempEMP1) = 20 pages , size (tempEMP2) = 40 pages
size (tempDEPT1) = 3 pages , size (tempDEPT2) = 5 pages
```

Figure 5 shows the QG graph built for these queries. The edge $q_1 \rightarrow q_2$ is introduced because tempDEPT2 can be derived from tempDEPT1, while the edge $q_2 \rightarrow q_1$ is introduced because tempEMP1 can be derived from tempEMP2. The cycle is broken by removing the edge $q_1 \rightarrow q_2$ for a total savings of 60 page accesses.

Although algorithm **BS** provides better plans than **AS** it still does not take advantage of all common subexpressions because of the requirement that queries must be run in some order and no interleaving is possible. In the next section we present another approach which takes advantage of all common subexpressions that can be identified in locally optimal plans.

6. Decomposition Algorithm

If query processing is done based on creating temporary intermediate relations, then it is known from existing algorithms [WONG76] that it is beneficial to break the query down to smaller and simpler subqueries. In the case of global query optimization, a similar approach seems promising also. Relaxing the assumption of the previous section which forced each plan to



$$\begin{tikzcd} q_1 \arrow[r, bend left, "5"] & q_2 \arrow[l, bend left, "60"] \end{tikzcd}$$

Figure 5: QG Graph for Queries Q_1 and Q_2

be processed totally before other plans start being processed, we will examine here the possibility of *interleaving* the execution of various access plans. Algorithm **D** (Decomposition) takes an approach based exactly on this idea of interleaved plan execution.

The main idea is to decompose the given queries into smaller subqueries and run those in some order depending on the various relationships among the queries. Then, the results of various subqueries are simply 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. As it was the case in the previous algorithms, we consider only locally optimal plans. A final assumption made for algorithm **D** is that temporary intermediate results are replacing relations used in tasks and this is done <u>without</u> changing the operations performed in the local plans. That is, the only transformation allowed is renaming of input relations. This restriction makes the global access plan produced by **D** easier to derive. Allowing more complex transformations on query plans in order to achieve even better utilization of temporary results is also possible and is described in the context of the heuristic algorithm of the following section.

Algorithm **D** proceeds as follows. First, as in **BS**, the queries that possibly overlap on some selections and joins are identified by checking the ground database relations that are used. For all queries $Q_i \in \mathbf{Q}$ that overlap with some other queries, we consider the corresponding plans P_i (local access plans) and define a directed graph $GP(V,E,L)$ (global access plan) in the following way

- $V = \bigcup_{i=1}^{n} V_i$
- $E = \bigcup_{i=1}^{n} E_i$
- For every $v_i \in V$, $L(v_i) = L_i(v_i)$

GP is in a sense the ***union*** of the local plans. We also define a function $Res : \mathbf{Q} \rightarrow V$ such that $Res(Q_i)=v_i$, where v_i is the node of plan P_i that provides the result to Q_i. Based on this graph, the decomposition algorithm performs some simple steps that introduce the effects of sharing among various tasks. The main idea is to avoid accessing the same data pages multiple times. Hence, the transformations that are done on the graph are based on changing the input relations to subqueries, to previously computed temporary relations. Figure 6 illustrates the basis of our transformations. In the following figures we use ***nemps*** for ***num_of_emps*** and ***dept*** for ***dept_name***. The temporary relation TEMP1 created by subquery SQ_1 can be further restricted to give the result of subquery SQ_2 ($SQ_2 \Longrightarrow SQ_1$). Therefore, TEMP1 can be used as the input to that last subquery, instead of EMP. This is accomplished by adding a new edge from the node representing SQ_1 to the corresponding node for SQ_2. Also the relation name in SQ_2 is changed to TEMP1.

Formally algorithm **D** proceeds as follows. After building the graph GP, the following transformations are performed in the order they are presented

[1] *Proper Implications* : Let $PI(v_i) = \{v_j \mid L(v_i) \Longrightarrow L(v_j) \text{ and } L(v_j) \not\Longrightarrow L(v_i)\}$. For a given task v_i, $PI(v_i)$ gives the set of tasks v_j, the results of which can be used by v_i as inputs instead of other base relations. Let $c_i \in PI(v_i)$ be the task such that $\forall\ v_j \in PI(v_i)$, $L(c_i) \Longrightarrow L(v_j)$ (if more than one such task exists, let c_i be the one belonging to the plan P_k with the least k). In other words c_i is the ***strongest*** condition that can be performed on some input relation(s) so that the result of this condition can still be used to answer v_i. Then, replace the occurrences of base relations used in tasks v_i with the corresponding temporary relations $TEMP_k$ found in the

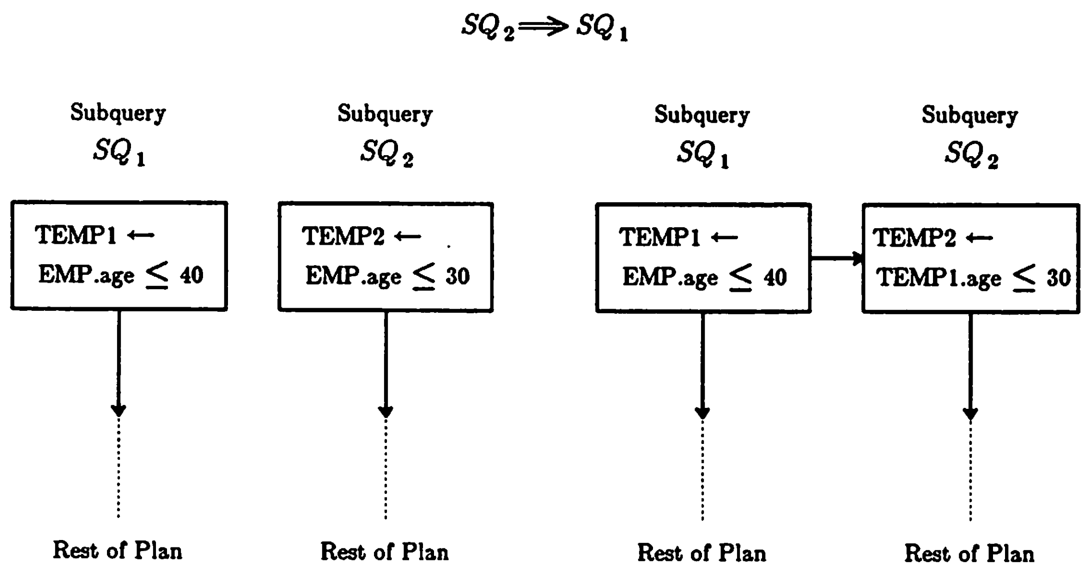


Figure 6: Basic Merge Operation

tasks c_i=[$TEMP_k \leftarrow expr$]. This is accomplished by adding an edge $c_i \rightarrow v_i$ and changing $L(v_i)$ by substituting the relation name involved in the selection or join to the name of the temporary relation which holds the result in $L(c_i)$ (i.e. $TEMP_k$).

[2] *Identical Nodes* : In the case of nodes that produce identical temporary relations a simple step is used to compute that temporary relation result only once and then change relation names to the one selected to hold the result. First, the equivalence classes C_i are identified, each composed of nodes from V, such that for every $v_j, v_k \in C_i$, $L(v_j) \equiv L(v_k)$. Select the vertex v_j belonging to the plan P_j with the least index j as the representative c_i of class C_i. Then, for each equivalence class C_i, remove from the graph GP all nodes $v_j \in C_i - \{c_i\}$ and substitute each edge $v_j \rightarrow v_k$ with a new edge $c_i \rightarrow v_k$. Let $L(v_k)$=[$TEMP_k \leftarrow expr_k 1$], for all such v_k. Also let c_i=[$TEMP_i \leftarrow expr_i$]. Change all occurrences of relation name $TEMP_k$ in v_k to $TEMP_i$. Finally, if for some query Q_m, $v_j = Res(Q_m)$ and $v_j \in C_i$, set $Res(Q_m)$ to c_i. This last step makes sure that identical final results are never computed more than once.

[3] *Recursive Elimination* : Because steps [1] and [2] may have introduced new nodes that are now identical, step [2] is repeatedly applied until it fails to produce any further reduction to the graph GP. Example of such a case is a join performed on two relations that are restricted with identical selection clauses. Step [2] will merge each pair of identical selections to a single one and 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 fact that any transformation performed on the graph in all cases adds new edges 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 $v_1 \Longrightarrow v_2 \Longrightarrow \cdots \Longrightarrow v_1$. 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 D suggests. The function Res also gives the nodes that hold the results for all queries.

To give an example of the algorithm, Figures 7, 8 and 9 show the initial access plan graphs, the graph GP after transformation [1] and the final global access plan graph (as a sequence of operations) respectively for the two queries Q_1 and Q_2 of section 3.

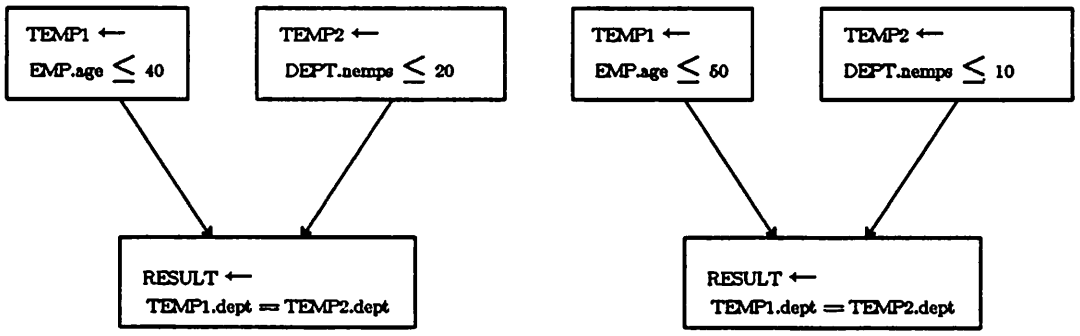


Figure 7: Initial Global Access Plan

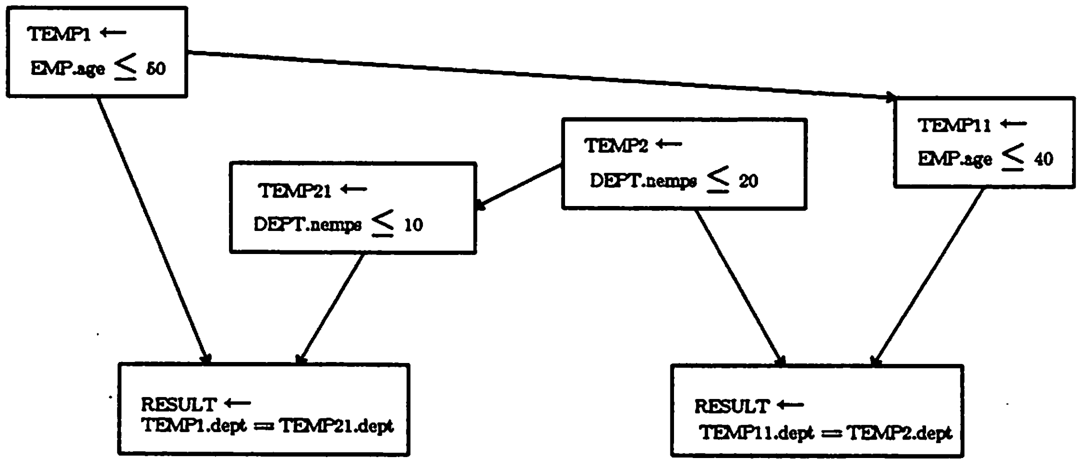


Figure 8: Global Access Plan after Transformation [1]

```
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
```

Figure 9: Final Global Access Plan

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

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

where CS is now the set of <u>all</u> common subexpressions found in the local access plans and n_s and $savings(s)$ are defined in the same way as in the previous section. For example, for the queries Q_1 and Q_2, $Cost(GP') = 223 + C_j(20,5) + C_j(40,3)$, where $C_j(a,b)$ is the cost function for a join between two relations as introduced in the previous section. This cost represents a savings of 65 page accesses compared to an arbitrary serial execution. Concerning the complexity of the algorithm, it can be observed that steps [1] and [2] of the above algorithm require time in the order of $\prod_{i=1}^{k} | V_i |$, 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 [2] 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 $N \cdot \prod_{i=1}^{k} | V_i |$.

We now move on to discuss the most 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).

7. Heuristic Algorithm

As it was illustrated through an example in section 4, merging locally optimal plans to produce the global access plan is not always the optimal strategy. The main reason is that there are more than one possible plans to process a query, yet the algorithms presented in the previous sections consider only one of them, i.e. the optimal in terms of execution time. Using suboptimal plans may prove to be better. Grant and Minker in [GRAN80] present a Branch and Bound algorithm [RICH83] that uses more than locally optimal plans. One assumption they make is that queries involve only equijoins while all selections are of the form $R.A{=}cons$. This section presents a similar algorithm which is defined as a state space search algorithm (A* [RICH83]) with better average case performance than the one of [GRAN80]. 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 predicates in queries are discussed.

As shown in Figure 1, the Global Optimizer receives as input a set of queries $\mathbf{Q}=\{Q_1,Q_2, \cdots ,Q_n\}$. Then for each query Q_i a set of possible plans that can be used to process that query is derived. Let that set be $\mathbf{S}_i=\{P_{i1}, P_{i2}, ..., P_{ik_i}\}$. For a given query Q_i, $\mathbf{S}_i$ contains the optimal plan to process Q_i along with all other possible plans that share tasks with plans for other queries. For example, for the two queries Q_3 and Q_4 of section 3, in addition to the plans P_3 and P_4 presented there the plan

```
(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
```

should also be considered for query Q_3 because it shares the join `EMP.dept_name=DEPT.dept_name` with P_4. Hence, the sets of plans $\mathbf{S}_3$ and $\mathbf{S}_4$ will be $\mathbf{S}_3=\{P_3,P_{32}\}$ and $\mathbf{S}_4=\{P_4\}$. Generally, this algorithm considers optimizing a set of queries instead of a set of plans, which was the case with algorithms **BS** and **D**. Considering more than one candidate plans per query has the desirable effect of detecting and using effectively all common subexpressions found among the queries.

This section is organized as follows: in the first subsection a state space is defined and an A* algorithm that finds the solution by searching that space is described. Then subsection 7.2 presents a preprocessing step that can be applied in order to improve the average case performance of the algorithm. Finally, the last subsection discusses the performance of the algorithm and suggests some possible extensions.

7.1.1. The Heuristic Algorithm

In order to present an A* algorithm, one needs to define a state space $\mathbb{S}$, the way transitions are done between states and the costs of those transitions.

Definition 1 : A *state* s is an n-tuple $<p_1,p_2, \ldots ,p_n>$, where $p_i \in \{\text{NULL}\} \cup \mathbf{S}_i$. If $p_i = \text{NULL}$ it is assumed that state s suggests no plan for evaluating query Q_i.

Definition 2 : Let $s_1=<p_1,p_2, \ldots ,p_n>$ and a function *next* : $\mathbb{S} \rightarrow \mathbb{Z}$ with

$$next(s_1) = \min\{j \mid p_j=\text{NULL}\} \qquad \text{if } \{j \mid p_j=\text{NULL}\}\neq\emptyset$$

A *transition* $T(s_1,s_2)$ from state s_1 to s_2 exists iff s_1 has at least one NULL entry and $s_2=<q_1,q_2, \ldots ,q_n>$, with $q_i=p_i$ for $1 \leq i < next(s_1)$, $q_{next(s_1)} \in \mathbf{S}_{next(s_1)}$ and

q_j=NULL, for $next(s_1)+1 \leq j \leq n$.

Definition 3 : The <u>*cost*</u> *tcost*(t) of a transition $t=T(s_1,s_2)$ is defined as the *additional* cost needed to process the new plan q_m introduced at t (according to Definition 2), given the (intermediate or final) results of processing the plans of s_1.

From the above definition it can be seen that the way transitions are defined, 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 algorithm. The state s_0=<NULL,NULL, . . . ,NULL> is the initial state of the algorithm and the states $s_f=<p_1,p_2, \ldots ,p_n>$ with $p_i \neq$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. For brevity it will be assumed that each plan is an unordered set of tasks instead of a directed graph. In order for an A* algorithm to have fast convergence, a heuristic function h is introduced on states [RICH83]. This function is used to prune down the size of the search space that will be explored. Such a function $h : \mathbb{S} \rightarrow \mathbb{Z}$ was introduced in [GRAN80] in the following way : let $s=<p_1,p_2,...,p_n>$ be some state. Then

$$h(s)= \sum_{i=next(s)}^{n} \min_j \left[est\_cost(P_{ij}) - \sum_t n_t \cdot est\_cost(t)\right]$$

where t are common tasks found in plans already in s and n_t is the number of times task t appears in these plans. The function *est_cost* is defined on tasks as follows

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

where n_q is the number of queries the task t occurs in. The idea behind defining such a function is that the cost of a task is amortized among the various queries that will *probably* make use of it. For a plan p, it is assumed that

$$est\_cost(p) = \sum_{t \in p} est\_cost(t)$$

If it is true that $est\_cost(p) \leq Cost(p)$ then the convergence of the A* algorithm is guaranteed [RICH83]. Therefore, one significant issue is to define a correct function *est_cost*, "correct" meaning that it underestimates the actual cost. Let us give an example, also drawn from [GRAN80], which will motivate the discussion of the following subsection.

Two queries Q_1 and Q_2 are given along with their plans : P_{11}, P_{12}, P_{21}, P_{22}, P_{23}. We will use t_{ij}^k to indicate the k-th task of plan P_{ij}. The table below gives the costs for the tasks involved in each plan

Plan	Task Cost	Task Cost	Task Cost	Total
P_{11}	t_{11}^1 40	t_{11}^2 30	t_{11}^3 5	75
P_{12}	t_{12}^1 35	t_{12}^2 20		55
P_{21}	t_{21}^1 40	t_{21}^2 10	t_{21}^3 5	55
P_{22}	t_{22}^1 10	t_{22}^2 30	t_{22}^3 10	50
P_{23}	t_{23}^1 30	t_{23}^2 20		50

and the identical tasks are
$t_{11}^1 \equiv t_{21}^1$; $t_{11}^2 \equiv t_{22}^2$; $t_{12}^2 \equiv t_{23}^2$;
Given the actual task costs and the sets of identical tasks, the estimated costs (*est_cost*) for these tasks are

Task	t_{11}^1	t_{11}^2	t_{11}^3	t_{12}^1	t_{12}^2	t_{21}^2	t_{21}^3	t_{22}^1	t_{22}^3	t_{23}^1
Estimated Cost	20	15	5	35	10	10	5	10	10	30

and the estimated costs for the plans are,

Plan	P_{11}	P_{12}	P_{21}	P_{22}	P_{23}
Coalesced Cost	40	45	35	35	40

Based on the above numbers and the construction procedure outlined, Figure 10 shows the search space $\mathbb{S}$ along with the costs of transitions between states and estimated costs of going from intermediate to final states.

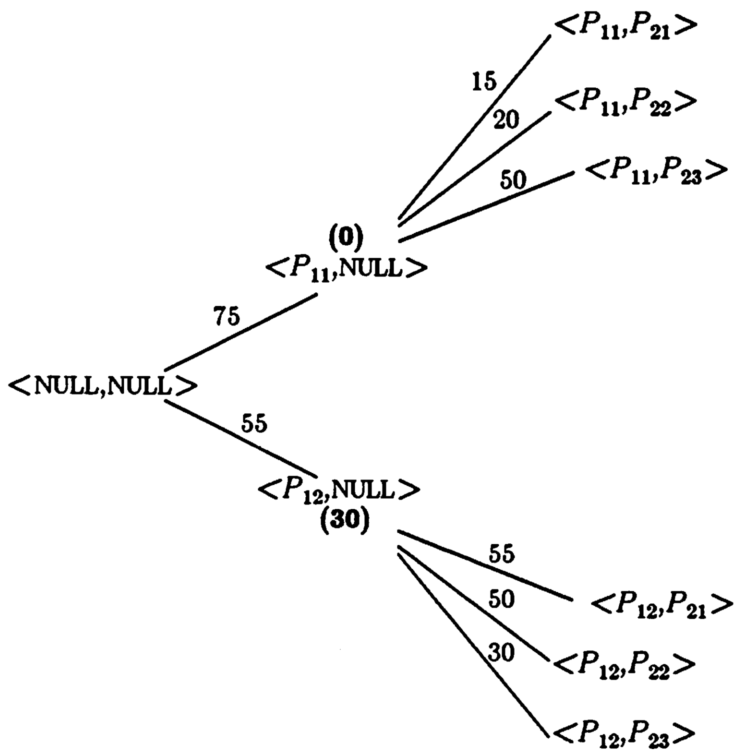


Figure 10: Example Search Space for A* Algorithm (numbers in parentheses show estimated costs)

Tracing the A* algorithm we get

$s_0 =$ <NULL,NULL>	/* *expand state* s_0 */
$s_1 =$ <P_{11},NULL>	/* *expand state* s_1 */
$s_2 =$ <P_{21},NULL>	/* *expand state* s_2 */
$s_F =$ <P_{12},P_{23}>	/* *the final solution* */

yielding $<P_{12},P_{23}>$ 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 behaviour 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_{11} will not be able to share both of its tasks t_{11}^1 and t_{11}^2 with plans P_{21} and P_{22} respectively, since only one of these two latter plans will be in the final solution (final state). Therefore, the value $est\_cost(P_{11})$ 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 the higher the estimator values the faster the convergence [RICH83]. Hence, estimating the cost function better will enable the algorithm to converge faster to the final solution.

7.1.2. The Modified Algorithm

The goal of this subsection is to describe a preprocessing phase which provides a way to compute a better cost estimation function. Suppose that n sets of plans $S_1, S_2, \ldots, S_n$ are given, with $S_i = \{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$. Assume also that the pairs of tasks $t_i \in P_{il}$ and $t_j \in P_{jm}$ such that $t_i \equiv t_j$ are known. We then define a directed graph $G(V,E)$ 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 i-th query, introduce a vertex v_{ij}
- For each pair $t_{kl}^i \in P_{kl}$, $t_{pq}^j \in P_{pq}$ of such identical tasks there is an edge connecting the two vertices $(v_{kl} \rightarrow v_{pq})$ if there is no other plan P_{pr} with a task t_{pr}^s such that $t_{pr}^s \equiv t_{pq}^j$

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 $G'(V',E')$ and let S' be the set of plans P_{ij} that have a corresponding vertex v_{ij} in G'.

What is achieved through that preprocessing phase, is to reduce considerably the size of the search space for the A* algorithm. Only plans in S' are considered in order to derive the *est_cost* values. To give an example of the preprocessing phase along with a run of the A* algorithm, we will redo the example of the previous subsection.

We are given again the same two queries and five plans : $P_{11}, P_{12}, P_{21}, P_{22}, P_{23}$. The graph of Figure 11 gives the graph G for the set of plans given.

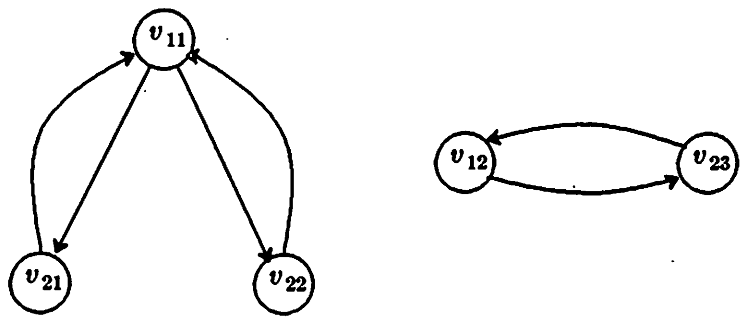


Figure 11: Graph G for Queries Q_1 and Q_2

After the DFS s performed the *Need* sets for the various vertices, will be

Vertex	*Need*
v_{11}	$\{11,21,22\}$
v_{12}	$\{12,23\}$
v_{21}	$\{11,21\}$
v_{22}	$\{11,22\}$
v_{23}	$\{12,23\}$

From the above table it can be seen that vertex v_{11} must be eliminated since it can reach both 21 and 22 through directed paths. After that, the edges $(v_{11}\rightarrow v_{21})$, $(v_{21}\rightarrow v_{11})$, $(v_{11}\rightarrow v_{22})$ and $(v_{22}\rightarrow v_{11})$ are marked as OUT. This causes vertices v_{21} and v_{22} to be deleted also. Finally, we see that no more vertices can be deleted. The remaining graph is shown in Figure 12.

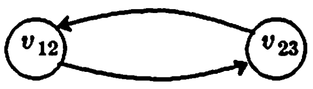


Figure 12: Final Graph G'

Finally, $\mathbf{S}' = \{P_{12}, P_{23}\}$.

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 [GRAN80] (which is identical to the estimator used in the previous subsection) is defined

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

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

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

Now, given a plan P_{ij} and a specific task t_{ij}^k, let Q_{ij} be the set of, other than i, queries q that have common tasks with P_{ij}. Also, let n_{ij}^q be the number of plans P_{qr} that correspond to query q in Q_{ij}. Then, *est_cost* is defined as follows

a) If the plan P_{ij} is not in $\mathbf{S}'$ and $n_{ij}^q > 1$ for at least one query q, then

$$est\_cost(P_{ij}) = Cost(P_{ij}) \; - \sum_{q \in Q_{ij}} \max\left[coalesced\_cost(t_{ij}^k)\right]$$

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

b) If the plan is in $\mathbf{S}'$ or it is not in $\mathbf{S}'$ but the above condition on n_{ij}^q is not true, then

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

Finally, we show how to compute the function $h(s)$. First, define

$$add\_cost(i) = \min_j \left[est\_cost(P_{ij}) \; - \sum_{t \in O_i \cap P_{ij}} n_t \cdot est\_cost(t)\right]$$

where $O_i = \bigcup_{l=1}^{i-1} P_{lk}$ and P_{lk} is the plan that provides, for a given l, the above minimum value in the computation of $add\_cost(l)$. Also, t are common tasks that belong to plans already in the state s and n_t is the number of times task t appears in these plans. Then, define

$$h(s) = \sum_{i=next(s)}^{n} add\_cost(i)$$

The A* algorithm can then be applied using these new estimators. For example, processing the two queries Q_1 and Q_2 given above, the following are the computed estimated costs for the plans

Plan	P_{11}	P_{12}	P_{21}	P_{22}	P_{23}
Estimated Cost	55	45	35	35	40

Tracing the A* algorithm, we see that it explores the following states

$s_0 =$ <NULL,NULL> /* *expand state* s_0 */
$s_1 = <P_{12}$,NULL> /* *expand state* s_1 */
$s_F = <P_{12},P_{23}>$ /* *the final solution* */

yielding again $<P_{12},P_{23}>$ as the optimal solution with cost 85. Notice that if the commands were executed sequentially it would have costed $Cost(P_{12}) + Cost(P_{23}) = 105$. 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.

Summarizing, the final algorithm is the following

ALGORITHM HA

[1]. Build graph G and apply the preprocessing DFS algorithm

[2]. For all queries with no representative plan in the initial graph G, find the originally cheapest plan and put it in the final solution

[3]. Based on the result set S', compute the function *est_cost*

[4]. For the rest of the queries run the A* algorithm described in the previous subsection

7.1.3. Discussion and Extensions

The global access plan is derived from integrating the local plans found in the final state s_F returned by the A* algorithm. The integrating process is very similar to the one described for the decomposition algorithm where local plan graphs are merged together. Examining the estimated cost of the global access plan, we have

$$Cost(GP) = \sum_{p \in s_F} Cost(p) - \sum_{s \in CS} n_s \cdot savings(s)$$

where CS represents the total number of subexpressions found in the n *queries* (not *plans* as it was the case in algorithm **D**) and n_s and $savings(s)$ are defined in section 5. Regarding the complexity of the algorithm **HA** we must notice that it is very hard to analyze the behaviour 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 cost estimation function is to the actual cost. However, the A* algorithm with the new estimator function we proposed will not take more steps than the original A* algorithm presented in subsection 7.2 (which uses *coalesced_cost* as its estimator function). This is based on the fact that for any task t it is true that $est\_cost(t) \geq coalesced\_cost(t)$. Therefore with the help of a known theorem [RICH83] our algorithm will give a solution in at most the same number of steps as the algorithm of [GRAN80].

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 to 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 before done through the use of $next(s)$, but rather any available (NULL) position of s. This results to larger fanout for each state and clearly more processing for the A* algorithm. The heuristic 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 $t_i \Longrightarrow t_j$ and $t_j \not\Longrightarrow t_i$ must be considered as well.

8. Some Experimental Results

We expect that for a large number of applications and query environments global 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 [RTI84] and the version of INGRES that is commercially available. The experiments were run over the set of queries that Finkelstein used in [FINK82]. 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 the Appendix. Seven different sets of queries QSET1-QSET7 where chosen and the queries within each of these sets were processed

a) as independent queries

b) as the Better Serial Execution Algorithm suggests

c) as the Decomposition Algorithm suggests, and finally

d) as the Heuristic Algorithm suggests.

Table 1 describes some characteristics of the sets QSET1 to QSET7.

Query Set	Number of Queries	Queries	BS	D	HA
QSET1	2	{1,7}	X		
QSET2	2	{1,6}	X		
QSET3	4	{1,2,6,7}	X		
QSET4	2	{6,7}	X		
QSET5	4	{2,3,4,6}	X	X	
QSET6	7	{1,2,3,4,5,6,7}	X	X	
QSET7	2	{7,8}	X	X	X

Table 1: Query Sets Used in Experiments

The second column indicates the number of queries used in each set while the third column shows which queries from Appendix A were specifically used. The rest three columns indicate which algorithms were applicable to each of the given query sets. In general, not all algorithms give distinct global access plans. For example, in section 5.2 it was shown that if the query graph *QG* is acyclic, algorithms **BS** and **D** will give the same result.

The above sets of queries were tested in various settings. First, unstructured relations were used with their sizes varied according to Table 2.

Relation	Number of tuples
Employees	100 - 10,000
Corporations	10 - 500
Schools	20 *(fixed)*

Table 2: Sizes of relations

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)
```

Finally, in a 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. As it was indicated in section 5.2, the formula that provided an estimate on the cost savings using a global optimization algorithm is (for n queries $Q_1, ..., Q_n$)

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

where CS is the set of common temporary results s and n_s is the number of queries using the same temporary result s. 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 n_s affected the cost of processing the global access plans.

The measure used in this performance analysis was

$$PERCI = \frac{Cost_1(I/O) - Cost_2(I/O)}{Cost_1(I/O)} \qquad \text{(F)}$$

where $Cost_1(I/O)$ is the number of I/O's required to process all queries assuming no global optimization is performed. $Cost_2(I/O)$ is the corresponding figure in the case where a global access plan is constructed according to some of the presented optimization algorithms. 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.

8.1.1. Unstructured Relations

As indicated in Table 1, some query sets were processed using only one or two of the algorithms. Because of the similarity of the results we will group the diagrams according to the algorithm used for optimization. Hence, three diagrams are presented. One for query sets QSET1, QSET2 and QSET3, one for QSET4, QSET5 and QSET6 and another for QSET7. The first group was optimized using only **BS** because **D** and **HA** were not applicable. The second group was optimized using **BS** and **D** while for the last group all three algorithms were used. Figures 13, 14 and 15 illustrate how *PERCI* varies for the three above mentioned groups according to the size of the database in the case of unstructured relations. Also, Figure 16 gives the overall average improvement in the performance of the system for all query sets. 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, it is clear that there is always a gain in performance by doing global query optimization, i.e. *PERCI*$\geq$0 in all the above figures. Second, after some size of the relations, *PERCI* starts to decrease. This was due to the specific

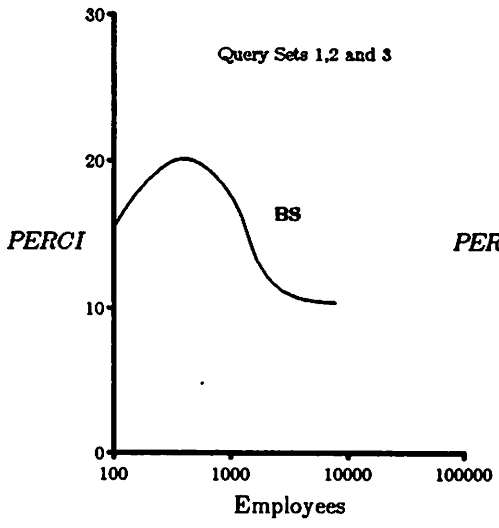


Figure 13

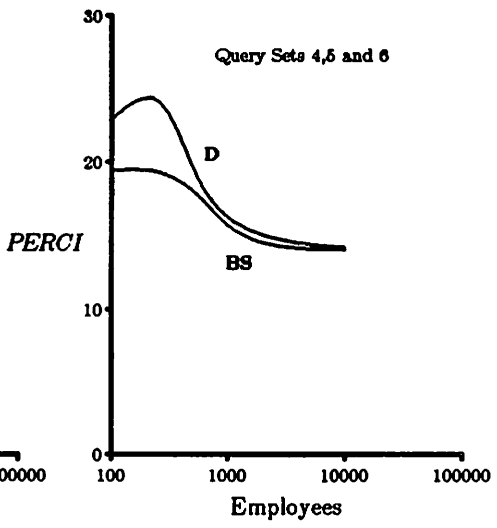


Figure 14

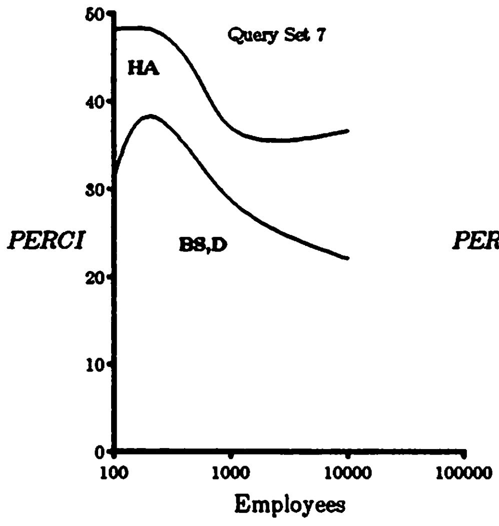


Figure 15

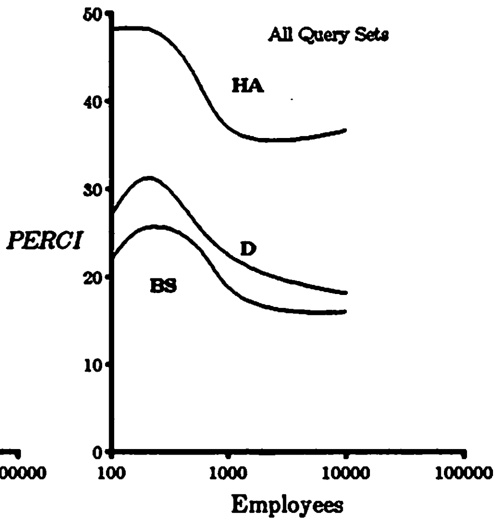


Figure 16

Performance Improvement for Unstructured Relations

type of queries used. In particular, because of queries involving joins, the denominator of the formula (F) 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 which 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.

The diagrams also show that there was no significant difference between the improvements achieved by the **BS** and **D** algorithms. In order to have a difference in the global plans generated by the two algorithms, as discussed in section 6, cycles must occur in the query graph QG. Even in that case though, the difference may not be significant depending on the sizes of the temporary results. In the experiments ran, the temporary relations not shared by more than one queries in the global access plan constructed by **BS** but shared in the corresponding plan generated by **D**, were rather small. Hence, sharing of these relations contributed only marginally to the performance improvement. Finally, for the last query set QSET7, the plan generated by **HA** was significantly better than the one generated by **BS** (or **D** since these are the same for QSET7). 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.

8.1.2. 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 17, 18 and 19 illustrate how *PERCI* varies for the three above mentioned groups according to the size of the database in the case of structured relations. Also, Figure 20 gives again the overall average improvement in the performance of the system for all query sets. Comparing the values of *PERCI* with the corresponding ones of the previous subsection, we can

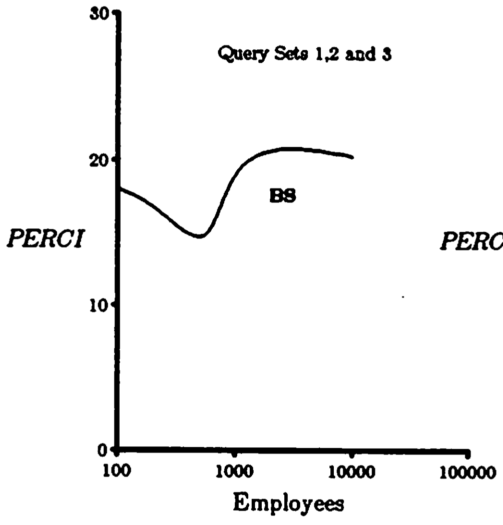


Figure 17

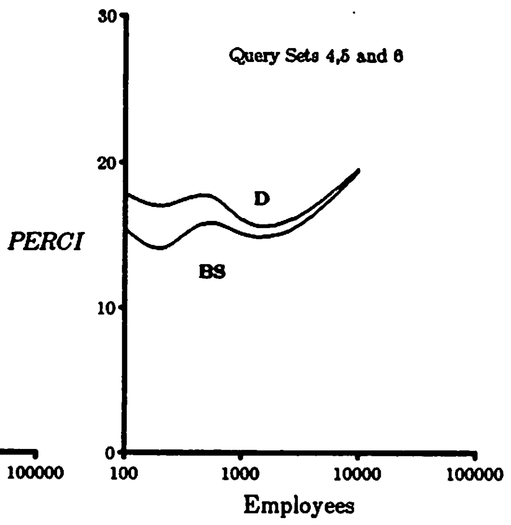


Figure 18

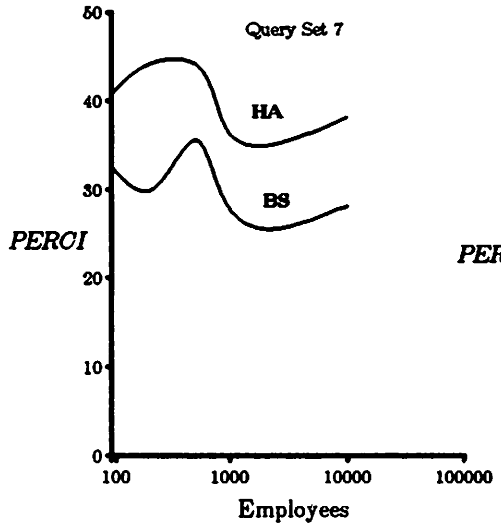


Figure 19

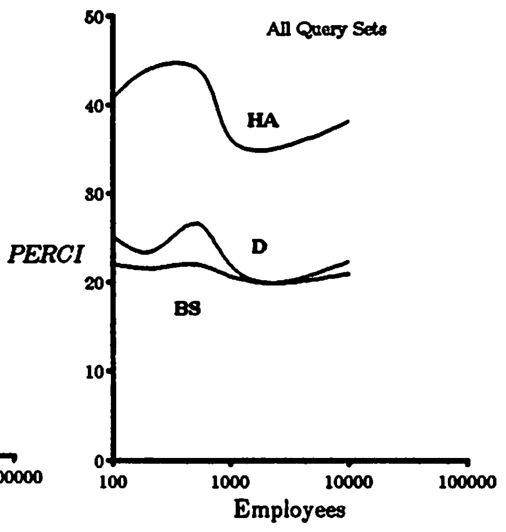


Figure 20

Performance Improvement for Structured Relations

observe some decrease of 10-20% for **BS**, **D** 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 Employee, *PERCI* starts increasing instead of decreasing, which was the case in the experiments of the previous subsection. This behaviour 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 relation Employee *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 the other two and **D** provides better plans than **BS**.

8.1.3. 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 $\text{experience} \geq 20$ found in queries 2,4,5 and 7 to $\text{experience} \geq 10$. 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 21 illustrates how *PERCI* varied with the size of the database in the case of unstructured relations and for the second group of query sets (i.e. QSET4, QSET5 and QSET6).

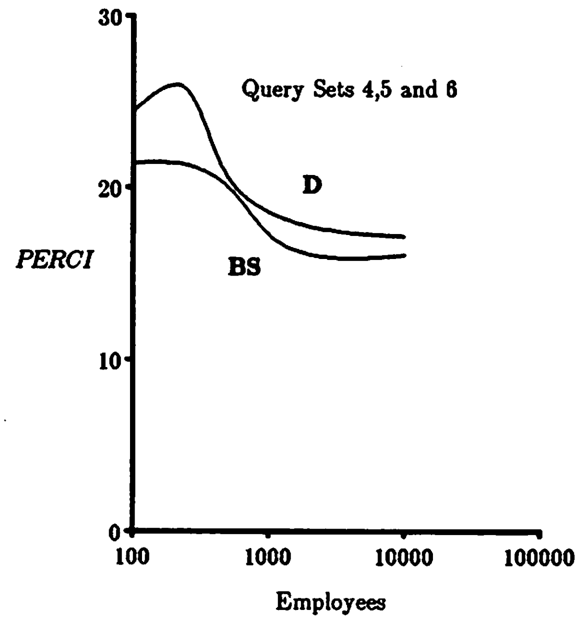


Figure 21: Performance Improvement for Higher Sharing

The rest of the query sets were 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 14. However, because of the higher degree of sharing among queries an increase of about 10% in the performance improvement was observed.

9. Summary

This paper presented a set of algorithms that can be used for multiple query processing. The main motivation for doing such interquery analysis is the fact that common intermediate results may be shared among various queries. We showed that various algorithms can be used for global query optimization. These algorithms were presented as parts of an algorithm hierarchy; as we descend the hierarchy more sophisticated algorithms can be used that 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 isolating common subexpressions. The last (heuristic search) algorithm, is a variation of the algorithm for optimizing a set of relational expressions originally proposed by Grant and Minker in [GRAN80]. The preprocessing phase added to the algorithm intends to derive a better cost estimator function used in the A* algorithm.

We expect that for a large number of applications and query environments global query optimization will offer substantial improvement to the performance of the system. 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 at least 20-50% in both I/O and CPU time. We should also mention that our methods do not pose any problems to the concurrency control and recovery modules. Since the given set of queries is thought as a transaction itself, changing the way processing is done has no effect on the system. The transaction boundaries are preserved. In terms of concurrent access, it should also be clear that our transformations do not affect the degree of concurrency. The data that each query processes is exactly the same as in any arbitrary serial execution of the queries. Hence, we neither increase nor decrease the size of the data sets that each query competes for.

As interesting future research directions in the area of global 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. Also the application of our method in rule-based systems in general seems like a very interesting problem for investigation. For example, PROLOG and database systems based on logic [ULLM85] can easily be extended to perform global query optimization. Finally, some of the techniques that we developed here, can be applied in processing recursion in database environments [IOAN86]. 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.

10. REFERENECES

[ASTR76] Astrahan, M. et al, *"System R: A Relational Approach to Database Management"*, ACM Transactions on Database Systems, (1) 2, June 1976.

[CHAK82] Chakravarthy, U.S. and Minker, J., *"Processing Multiple Queries in Database Systems"*, in Database Engineering , (1), 1983.

[CHAK84] Chakravarthy, U.S., Fishman, D.H. and Minker, J., *"Semantic Query Optimization in Expert Systems and Database Systems"*, in [KERS84].

[CHAK85] Chakravarthy, U.S. and Minker, J., *"Multiple Query Processing in Deductive Databases"*, University of Maryland, Technical Report TR-1554, College Park, MD, August 1985.

[CHAK86] Chakravarthy, U.S., Minker, J. and Grant, J. *"Semantic Query Optimization: Additional Constraints and Control Strategies"*, in [KERS86].

[CLOC81] Clocksin, W. and Mellish, C., Programming in PROLOG , Springer-Verlag, New York, NY, 1981.

[FINK82] Finkelstein, S., *"Common Expression Analysis in Database Applications"*, Proceedings of the 1982 ACM-SIGMOD International Conference on the Management of Data, Orlando, FL, June 1982.

[GALL78] Gallaire, H. and Minker, J., Logic and Data Bases , Plenum Press, New York, 1978.

[GARE79] Garey, M.R. and Johnson, D.S., Computers and Intractability , W.H. Freeman and Co, San Francisco 1979.

[GRAN80] Grant, J. and Minker, J., *"On Optimizing the Evaluation of a Set of Expressions"*, University of Maryland, Technical Report TR-916, College Park, MD, July 1980.

[GRAN81] 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.

[GUTT84] Guttman, A., *"New Features for Relational Database Systems to Support CAD Applications"*, PhD Thesis, University of California, Berkeley, June 1984.

[HALL74] Hall, P.V., *"Common Subexpression Identification in General Algebraic Systems"*, IBM United Kingdom Scientific Centre, Technical Report UKSC 0060, November 1974.

[HALL76] Hall, P.V., *"Optimization of a Single Relational Expression in a Relational Data Base System"*, IBM Journal of Research and Development, **(20)** 3, May 1976.

[IOAN86] Ioannidis, Y., *"Processing Recursion in Deductive Database Systems"*, PhD Thesis, University of California, Berkeley, July 1986.

[JARK84a] Jarke, M., Clifford, J. and Vassiliou, Y., *"An Optimizing PROLOG Front-end to a Relational Query System"*, Proceedings of the 1984 ACM-SIGMOD International Conference on the Management of Data, Boston, MA, June 1984.

[JARK84b] 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.

[KIM84] 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.

[KUNG84] Kung, R. et al, *"Heuristic Search in Data Base Systems"*, in [KERS84].

[LARS85] Larson, P. and Yang, H., *"Computing Queries from Derived Relations"*, Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.

[NAQV84] Naqvi, S. and Henschen, L., *"On Compiling Queries in Recursive First-Order Databases"*, Journal of the ACM, (31) 1, January 1984.

[RICH83] Rich, E., Artificial Intelligence , McGraw-Hill, 1983.

[ROSE80] Rosenkrantz, D.J. and Hunt, H.B., *"Processing Conjunctive Predicates and Queries"*, Proceedings of the 6th International Conference on Very Large Data Bases, Montreal, October 1980.

[ROUS82a] Roussopoulos, N., *"View Indexing in Relational Databases"*, ACM Transactions on Database Systems, (7) 2, June 1982.

[ROUS82b] Roussopoulos, N., *"The Logical Access Path Schema of a Database"*, IEEE Transactions on Software Engineering, (8) 6, November 1982.

[RTI84] EQUEL/C User's Guide , Version 2.1, Relational Technology, Inc., Berkeley, CA, July 1984.

[SELL85] Sellis, T. and Shapiro, L., *"Optimization of Extended Database Languages"*, Proceedings of the 1985 ACM-SIGMOD International Conference on the Management of Data, Austin, TX, May 1985.

[SELL86] Sellis, T., *"Global Query Optimization"*, Proceedings of the 1986 ACM-SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[STON76] Stonebraker, M. et al, *"The Design and Implementation of INGRES"*, ACM Transactions on Database Systems, (1) 3, September 1976.

[STON86] Stonebraker, M. and Rowe, L., *"The Design of POSTGRES"*, Proceedings of the 1986 ACM-SIGMOD International Conference on the Management of Data, Washington, DC, May 1986.

[ULLM82] Ullman, J., Principles of Database Systems , Computer Science Press, 1982.

[ULLM85] Ullman, J., *"Implementation of Logical Query Languages for Data Bases"*, Proceedings of the 1985 ACM-SIGMOD International Conference on the Management of Data, Austin, TX, May 1985.

[WONG76] Wong, E. and Youssefi K., *"Decomposition: A Strategy for Query Processing"*, ACM Transactions on Database Systems, (1) 3, September 1976.

[ZANI83] Zaniolo, C., *"The Database Language GEM"*, Proceedings of the 1983 ACM-SIGMOD International Conference on the Management of Data, San Jose, CA, May 1983.

APPENDIX

The set of queries used in the experiments of section 8 were the following

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

range of e is Employees
range of c is Corporations
range of c1 is Corporations
range of s is Schools
```

```
/* get all employees with more than 10 years experience */
(1)  retrieve (e.all) where e.experience ≥ 10

/* get all employees less than 65 years old with more than 20 years
   experience */
(2)  retrieve (e.all) where e.experience ≥ 20 and e.age ≤ 65

/* get all pairs (employee,corporation), where the employee
   has more than 10 years experience and works in a corporation with
   earnings more than 500K and located anywhere but in Kansas */
(3)  retrieve (e.all,c.all)
   where e.experience ≥ 10 and e.employer=c.cname
   and c.location ≠ ''KANSAS'' and c.earnings > 500

/* get all pairs (employee,corporation), where the employee
   has more than 20 years experience and works in a corporation with
   earnings more than 300K and located anywhere but in Kansas */
(4)  retrieve (e.all,c.all)
   where e.experience ≥ 20 and e.employer=c.cname
   and c.location ≠ ''KANSAS'' and c.earnings > 300

/* get all pairs (president,corporation), where the president
   is less than 65 years old with more than 20 years experience and the
   corporation is located in NEW YORK and has earnings more than 500K */
(5)  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

/* get all pairs (president,corporation), where the president
   is less than 60 years old with more than 30 years experience and the
   corporation is located in NEW YORK and has earnings more than 300K */
(6)  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

/* get all triples (employee,corporation,school) where the employee
   is less than 65 years old, has more than 20 years experience and holds
   a university degree working for a corporation located in NEW YORK and
   with earnings more than 500K */
(7)  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''
```

/ get all pairs (employee,corporation), where the employee*
is less than 65 years old with more than 20 years experience and the
*corporation is located in NEW YORK and has earnings more than 300K */*

```
(8)  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
```