GLOBAL QUERY OPTIMIZATION †

Timos K. Sellis ††
University of California, Berkeley

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

†† Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720
{seismo,decvax,ihnp4}!ucbvax!ucbingres!timos
timos@ingres berkeley edu

† 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 Collections of queries can also be produced, for example, in the course of answering a query in a rule based (or deductive database) system [GALL78] In [SELL85a] 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)

Given a set of queries, it is a common practice to optimize each query separately To "optimize" a query means to choose among the various ways of executing the query 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 However, given a set of queries, there may be some common tasks that are found in more than one of them. Taking advantage of these common tasks, mainly by avoiding redundant page accesses, may prove to have a considerable effect on execution time

The problem of multiple query processing has been examined in the past Hall, for example, uses heuristics to identify common

subexpressions, especially within a single query [HALL74,HALL76] He uses a bottom-up procedure to identify common parts in the operator tree that is used to represent a query 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 information on the access structures that can be used in order to evaluate the set of queries Then, at the second stage the "Optimizer" groups queries and executes them separately as groups instead of one at a time However, the algorithm that chooses the best access plan for each individual query suffers from very bad worst case behaviour 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) Other researchers have also recently examined the problem of global optimization Chakravarthy and Minker [CHAK82,CHAK85] propose a global query optimization algorithm based on the construction of an integrated query graph Their algorithm is a generalization of the query decomposition algorithm of [WONG76] but it does not guarantee that the access plan constructed is the cheapest one Kim in [KIM84] suggests also a two stage optimization procedure similar to the one in [GRAN81] The unit of sharing in Kim's proposal is the relation which is not always the best thing to do, 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 queries, is also related to the problem of multiple query optimization since it provides algorithms to check for common subexpressions Finally, Jarke in [JARK84b] presents the problem of common subexpression isolation in multiple query optimization He describes how common expressions can be detected and used according to their type (e g single relation restrictions, joins, etc)

Our approach to the problem of multiple query processing is based on the extend to which existing optimizers can be used without severe alterations Figure 1 illustrates the alternative architectures that can be used for multiple query processing

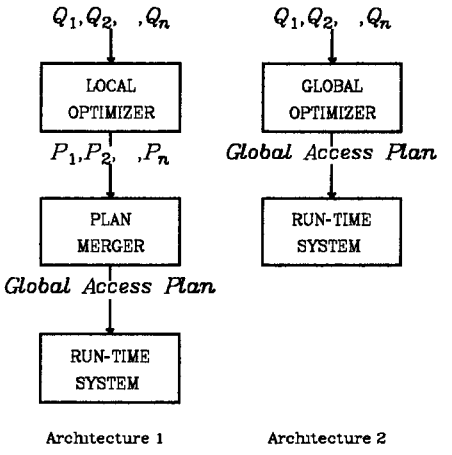

Figure 1

Architecture 1 can be used with minimal changes to existing optimizers The *Plan Merger* is a component that uses the locally optimal plans generated by a conventional *Local Optimizer* and transforms them into a larger plan, the global access plan, which is in turn processed by the *Run-Time System* Since in many existing systems queries are compiled and saved as object code, it is very interesting, given a set of such pieces to identify a sequence in which they must be run to reduce the I/O and/or CPU cost Looking at more sophisticated procedures which, for example, reduce the cost of doing multiple joins involving the same relation R by scanning R once and examining several conditions in parallel [CHAK85], is also an acceptable solution Using such a procedure implies rewritting the query processor However, the redeeming value of such an effort cannot be justified because the outcome is not always the optimal plan To include such procedures in our framework, 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 thereby is not restricted to using locally optimal plans The purpose of this paper is to exhibit a hierarchy of optimization algorithms that can be used for multiple query optimization either as Plan Mergers or as Global Optimizers The algorithms

to be presented vary on the complexity of the Plan Merger and on whether Architecture 1 or 2 is used We also discuss the trade offs between the complexity of the algorithms and the optimality of the global plan produced

The paper is organized as follows In the next section we define the query model that we will use throughout this paper and present a formulation for the global query optimization problem Then, in section 3 we present our approach to the problem and introduce through the use of some examples some algorithms that can be used to solve the multiple query optimization problem Sections 4 through 6 present these algorithms in more detail Section 4 suggests an algorithm which finds a serial sequence for executing the queries which has better performance than an arbitrary serial execution Then, in section 5 we describe an algorithm that decomposes queries into smaller ones and produces a global access query plan Section 6 defines a state model and then proposes an A* algorithm similar to the one proposed by Grant and Minker with an extra preprocessing step that improves performance Finally, section 7 summarizes our results and suggests future directions for research in multiple query optimization

2 FORMULATION OF THE PROBLEM

We assume that we are given a database **D** as a set of relations $\{R_1,R_2, \ ,R_m\}$, and a set of queries $\mathbf{Q} = \{Q_1,Q_2, \ ,Q_n\}$ on **D** 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 share the following properties

[1] They are conjunctions of selection and join predicates

[2] They do not have projection lists

Under the above restrictions we exclude aggregates as well as predicates of the form $R_1\,A+R_2\,B=R_3\,C$ Extending a system to support these predicates is possible but would require significant increase in its complexity The restriction on conjunctive queries is one that is imposed by most of the papers in the literature and is based on the fact that the result of a disjunctive query can really be thought of as the union of the results of the disjuncts Also equijoins are the ones mostly used in queries and therefore it is natural to examine this case only Finally, not allowing projections enables us to concentrate on the problem of using effectively the results of common subexpressions rather than the problem of deriving the result of a query from the result of another query ([LARS85] and [FINK82] provide some ideas in the context of reusing results of queries with projection lists)

A *task* is an expression *relname* ← *expr*, where *relname* is a name of a temporary relation or the keyword *RESULT*, indicating that this task provides the result of the query, and *expr* is either a conjunction of selection predicates over the same relation or a join between two possibly restricted relations (this latter type covers queries which are processed not by performing selections first followed by a join, but in a way that supports ' pipelining") From now on we will refer to tasks as if they were simply the *expr* part, unless it is otherwise explicitly stated

We will say that a task t_i *implies* task t_j ($t_i \Rightarrow t_j$) iff t_i is a conjunction of selection predicates on a relation R and on attributes $A_1,A_2, \ ,A_k$ and 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

We will say that 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 the same relation and $E_1 \equiv E'_1$ and $E_2 \equiv E'_2$

An *access plan* for a query is a sequence of tasks that produces the result of the query Formally, an access plan is an acyclic directed graph $G=(V,E,L)$ (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* ← *expr*)

For example, consider the following query on the relations EMP(name,age,dept) and DEPT(dept,nemps) (we use QUEL [STON76] to express them)

```
retrieve (EMP all,DEPT all)
  where EMP age ≤ 40
  and DEPT nemps ≤ 20
  and EMP dept = DEPT dept
```

One way to process this query is

```
TEMP1  ← EMP age ≤ 40
TEMP2  ← DEPT nemps ≤ 20
RESULT ← TEMP1 dept = TEMP2 dept
```

The graph of Figure 2 shows the corresponding access plan

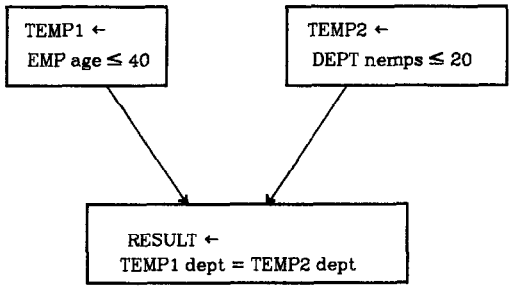

Figure 2

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 access plans in the obvious way

$cost(v_i)$ = the cost of I/O's needed

+ the CPU cost needed

for processing task t_i

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

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

We will refer to the minimal cost plans for processing each query Q_i individually, as *locally optimal* plans Respectively, 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 Generally the set of locally optimal plans and the globally optimal plan are different Finally, for a given query Q, $BCost(Q)$ gives the cost of the (locally) optimal plan P that can be used to evaluate Q Hence, $BCost(Q) = \min_{p \in \mathbf{P}}[Cost(p)]$, where **P** is the set of all possible plans that can be used to evaluate Q

Let us now consider a system that is given a set **Q** of queries and is required to execute it with minimal cost Given the above definition of an access plan, a global access plan is simply a directed labeled graph that provides a way to compute the results of all n queries Based on the above 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 (graphs) that can be used to process query i,*
>
> **Find** *a global access plan P by "merging" n local access plans (one out of each set $\mathbf{S}_i$) such that $Cost(P)$ is minimal*

The Plan Merger of Figure 1 performs the operation "merge" mentioned above It is the purpose of the following sections to define this operation and produce algorithms that find P

3 A HIERARCHY OF ALGORITHMS

Before we present our algorithms to find the optimal global access plan, we describe our basic approach to the problem The primary source of redundancy in multiple query processing is accessing the same data multiple times in different queries Our algorithms must recognize common subexpressions not to the extend of doing a thorough theorem proving but simply by isolating pairs of expressions e_1 and e_2 and checking if $e_1 \Rightarrow e_2$ based only on the form of the expressions and without going to the actual data stored in the database For example, e_1 may be EMP age ≤ 30 and e_2 may be EMP age ≤ 40 We do not consider cases like e_2 being EMP dept = "shoe" and it happens that all employees in the shoe department are under 40 years old because

of the specific instance of the database Unless such a rule is explicitly known to the system in the form of an integrity constraint or functional dependency, we cannot do any simplification to the query [JARK84a,CHAK84] Therefore, a significant part of our algorithms is devoted in common subexpression isolation

Second, it is the case in many systems that for some queries the optimal local access plans have been found and stored in the database (e g System-R [ASTR76] and POSTGRES [STON85] choose to do so) Because the system cannot afford to store more than one plan for each query, it stores only the optimal access plans Then, in subsequent requests the system can use the precomputed plans along with the ones produced for the rest of the given queries and find a globally optimal access plan It is an interesting question to see how global optimization can be achieved based *only* on locally optimal plans

We group the various algorithms that can be used for global query optimization in a hierarchy shown in Figure 3 As we descend the hierarchy the complexity of the algorithm grows and the total cost for processing the queries 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 simply executes these plans in an arbitrary order This corresponds to

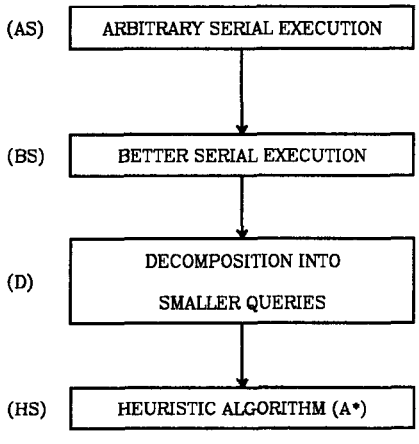

Figure 3

Architecture 1 of Figure 1 with the Plan Merger absent Algorithm BS preprocesses the plans and generates a better order of execution so that intermediate results (temporaries) are reusable using the algorithm of Finkelstein [FINK82] 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, etc

Algorithm D presents a different paradigm A query is decomposed into smaller parts and these parts become the unit of execution Therefore, a query is not processed as a whole but rather in small pieces which are assembled at various points to produce the result As an example why D might be a better algorithm than AS, consider the following database,

```
EMP (name,age,salary,job,dept)
DEPT (dept,nemps)
JOB (job,project)
```

without any fast access paths for any of the fields of the relations, and the queries

```
(Q1)  retrieve (EMP all,DEPT all)
        where EMP age ≤ 40
        and DEPT nemps ≤ 20
        and EMP dept = DEPT dept

(Q2)  retrieve (EMP all,DEPT all)
        where EMP age ≤ 50
        and DEPT nemps ≤ 10
        and EMP dept = DEPT dept
```

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 plan might be more efficient

```
retrieve into tempEMP (EMP all)
  where EMP age ≤ 50

retrieve into tempDEPT (DEPT all)
  where DEPT nemps ≤ 20

retrieve (tempEMP all,tempDEPT all)
  where tempEMP age ≤ 40
  and tempEMP dept = tempDEPT dept

retrieve (tempEMP all,tempDEPT all)
  where tempDEPT nemps ≤ 10
  and tempEMP dept = tempDEPT dept
```

because it avoids accessing the EMP and DEPT

relations more than once It is also significantly more efficient in the cases where restrictions reduce the size of the original relations a lot 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 results

Finally, algorithm HS is based on searching among local (not necessarily optimal) query plans and building a global access plan choosing one local plan per query Architecture 2 of Figure 1 applies to this case The effectiveness of algorithm HS is illustrated with the following example Suppose we have the queries

(Q_3) retrieve (JOB all,EMP all,DEPT all)
where EMP dept = DEPT dept
and JOB job = EMP job

(Q_4) retrieve (EMP all,DEPT all)
where EMP dept = DEPT dept

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 = DEPT dept

(P_4) retrieve (EMP all,DEPT all)
where EMP dept = DEPT dept

Notice that P_3 and P_4 do not share the common subexpression EMP dept=DEPT dept Algorithm HS considers in addition to P_3 the plan that processes the join clause EMP dept=DEPT dept 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

4 SERIAL EXECUTION

Algorithms AS and BS of Figure 3 are based on some serial execution of the given queries Q_1, Q_2, , Q_n As stated in the previous section we only consider the locally optimal plans P_i In the first case we do not impose any restrictions on the order in which the queries are processed which means that this is what a conventional query processor would do In the second case though some simple preprocessing can be done to achieve better performance We examine now in more detail these two algorithms

4 1 Arbitrary Serial Execution

In Algorithm AS the sequence in which the queries are run is chosen at random. We assume that all queries are processed without taking advantage of any common tasks that they may share The global plan P 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 $\mathbf{S} = \{Q_{i_1}, Q_{i_2}, \ldots, Q_{i_n}\}$, with $Q_{i_k} \in \mathbf{Q}$ and all, i_k distinct, the cost of the global access plan will be

$$Cost(P) = \sum_{j=1}^{n} BCost(Q_{i_j})$$

As an example, we consider the following queries Q_5 and Q_6

(Q_5) retrieve (EMP all)
where EMP age ≤ 40
and EMP salary ≤ 10

(Q_6) retrieve (EMP all)
where EMP age ≤ 40

Let us assume that the sizes of the initial relations and temporary results are as follows

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

We also assume that the local plans for Q_5 and Q_6 store temporaries for the above restrictions Then algorithm AS would require 110 page accesses for Q_5 and 120 page accesses for Q_6 Hence, the total cost would be 230 page accesses (throughout this paper we consider for simplicity only I/O costs in our examples)

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

4.2 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 various common subexpressions. The global access plan is again the concatenation of the local plans in the way described by **S**. Checking if a given temporary result can be used by a query is accomplished through the procedure that Finkelstein proposes in [FINK82]. It is our goal though not to increase significantly the complexity of the algorithm that derives the global access plan. This is the main reason that we consider *only* locally optimal plans. The algorithm of section 6 considers more local plans than simply the ones of minimal cost.

We now describe how algorithm BS works. We will build a directed graph that will eventually propose some execution order based on directed paths. This graph is very similar to the precedence graphs used in concurrency control [ULLM82]. First, we identify queries that are possibly overlapping on some expressions. 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 simply 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 $QG(V,E,L)$:

- For each plan $P_i(V_i,E_i,L_i)$ we define a node q_i.
- 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 $L_j(v_j) \Rightarrow L_i(v_i)$ and $L_i(v_i) \not\Rightarrow L_j(v_j)$.
  - 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 $i<j$.
- Let us assume that an edge $q_i \rightarrow q_j$ is introduced because of nodes v_i of P_i and v_j of P_j. 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 $L_i(v_i)$.

The second rule for edge definition is introduced to break ties in a specified manner. The resulting graph indicates how some intermediate result of one query can be used to compute the result of some other query. 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 then we break them by omitting a set of edges with minimal sum of labels and then produce **S** 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, it is the case that in multiple query optimization the graph will have just a few nodes and not many cycles, thus making the problem to have 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

$$Cost(P) = \sum_{i=1}^{n} BCost(Q_i) - \sum_{e \in E} L(e)$$

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

where CS is the set of common subexpressions s found in the queries and used in the final graph QG', n_s is the number of times the result of common subexpression s is used in the final sequence and $savings(s)$ 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 produce the result of s_1 and C_{s_1} be the cost of accessing the result of s_1 to compute the result of s_2. Then

$$savings(s) = \begin{cases} C_R - C_{s_1} & \text{for edges of type (a)} \\ C_R + C_{s_1} & \text{for edges of type (b)} \end{cases}$$

Example 1 Let us show with an example how BS works. We will use again the queries Q_5 and Q_6 of the previous subsection. The directed graph constructed is shown in Figure 4.

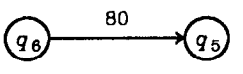

Figure 4

The edge $q_6 \rightarrow q_5$ is introduced because (EMP age $\leq$ 40 and EMP salary $\leq$ 10 $\Rightarrow$ EMP age $\leq$ 40 Therefore the serial execution will be S=$\{Q_6\ Q_5\}$ which uses 80 page accesses less than an arbitrary serial execution (230 page accesses), for savings of 35%

Example 2 To give an example where a cycle may occur, we consider queries Q_1 and Q_2 of section 3

(Q_1) retrieve (EMP all,DEPT all)
where EMP age $\leq$ 40
and DEPT nemps $\leq$ 20
and EMP dept = DEPT dept

(Q_2) retrieve (EMP all,DEPT all)
where EMP age $\leq$ 50
and DEPT nemps $\leq$ 10
and EMP dept = DEPT dept

with optimal local plans

(P_1) retrieve into tempEMP1(EMP all)
where EMP age $\leq$ 40

retrieve into tempDEPT1(DEPT all)
where DEPT nemps $\leq$ 20

retrieve (tempEMP1 all,tempDEPT1 all)
where tempEMP1 dept = tempDEPT1 dept

(P_2) retrieve into tempEMP2(EMP all)
where EMP age $\leq$ 50

retrieve into tempDEPT2(DEPT all)
where DEPT nemps $\leq$ 10

retrieve (tempEMP2 all,tempDEPT2 all)
where tempEMP2 dept = tempDEPT2 dept

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

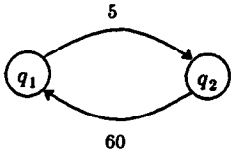

Figure 5

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 We break the cycle 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

5 DECOMPOSITION ALGORITHM

As it is known from conventional query processing [WONG76,SELI79], it is always beneficial to break the query down into smaller and simpler subqueries to avoid creating intermediate results of large sizes In the case of global query optimization, a similar approach seems promising also If we recognize subexpressions that are shared among queries, then instead of executing the queries in some serial order, we can do some decomposition into smaller queries, let these parts run in some order depending on the various relationships among the queries and then simply assemble the results of various subqueries to get the answers to the original queries This is exactly what our decomposition algorithm does The only restriction imposed is that the partial order defined in a local query plan must be preserved in the global query plan As it was the case in the previous algorithms, we consider only locally optimal plans Another assumption made for algorithm D is that we are allowed to use temporary results without changing the operations done in local plans, that is we are allowed to perform only simple renaming of relations This restriction makes the global access plan produced by D easier to derive and definitely better than an arbitrary serial execution In [SELL86] we examine a more general case for this algorithm (Complex Decomposition) which allows more complex transformations to query plans in order to achieve even better utilization of temporary results

Let us now present how algorithm D works First, we identify as in BS the queries that possibly overlap on some selections and joins by checking the original database relations that are used For each set of overlapping queries Q_i, we consider the corresponding plans P_i (local access plans), and define a directed graph $P(V,E,L)$ (global access plan) in the following way

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

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 of query Q_i Then we do the following transformations on P in the order they are presented

[1] *Proper Implications* Let $PI(v_i) = \{v_j \mid L(v_j) \Rightarrow L(v_i)$ and $L(v_i) \not\Rightarrow L(v_j)\}$ For every $v_j \in PI(v_i)$, add an edge $v_i \rightarrow v_j$ and change $L(v_j)$ by changing the relation name involved in the selection or join to the name of the temporary relation which holds the result in $L(v_i)$

[2] *Identical Nodes* Find the equivalence classes C_i, each composed of nodes from V, such that for every $v_j, v_k \in C_i$, $L(v_j) = L(v_k)$ 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 P all nodes $v_j \in C_i - \{c_i\}$ and substitute each edge $v_j \rightarrow v_k$ with a new edge $c_i \rightarrow v_k$ Also, change in $L(v_k)$ of all such v_k, the temporary relation names derived in v_j with the temporary relation names derived in c_i (substitute relation name) Finally, if for some $v_j \in C_i$, $v_j = Res(Q_m)$, set $Res(Q_m)$ to c_i

[3] *Recursive Elimination* Apply step [2] until it fails to produce any further reduction to the graph

The result of the above transformation is a directed graph P which is guaranteed to be acyclic if the initial graphs P_i are acyclic The directed arcs impose an order on the execution of the various tasks Finally, the function *Res* gives the nodes that hold the results for all queries To give an example of the algorithm, Figures 6, 7 and 8 show the initial access plan graphs, the graph P 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

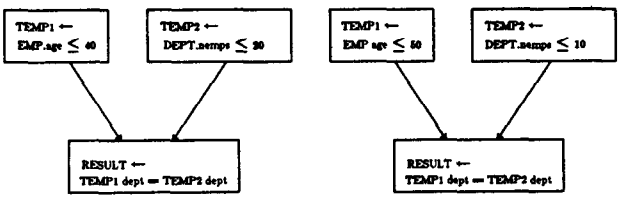

Figure 6

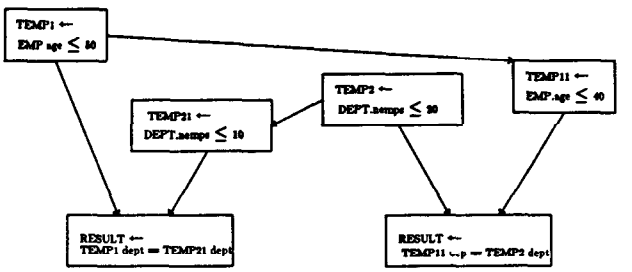

Figure 7

retrieve into TEMP1 (EMP all)
where EMP age ≤ 50

retrieve into TEMP2 (DEPT all)
where DEPT nemps ≤ 20

retrieve into TEMP11 (TEMP1 all)
where TEMP1 age ≤ 40

retrieve into TEMP21 (TEMP2 all)
where TEMP2 nemps ≤ 10

retrieve (TEMP11 all,TEMP2 all)
where TEMP11 dept = TEMP2 dept

retrieve (TEMP1 all,TEMP21 all)
where TEMP1 dept = TEMP21 dept

Figure 8

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

$$Cost(P) = \sum_{i=1}^{n} BCost(Q_i) - \sum_{s \in CS} n_s \; savings(s)$$

where CS is now the set of *all* 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, $Cost(P)$ = 223 + $CJ(20,5)$ + $CJ(40,3)$, where $CJ(a,b)$ is a cost function that gives the cost of doing a join between two relations of sizes a and b pages This cost represents a savings of 65 page accesses compared to an arbitrary serial

execution

We now move on to discuss the most general algorithm that can be used to process multiple queries This heuristic algorithm is the basis of the Global Optimizer module of Figure 1

6 HEURISTIC ALGORITHM

As it was illustrated through an example in section 3, it is not always the case that the locally optimal plans will be the best ones to integrate in order to get a globally optimal access plan Grant and Minker in [GRAN80] present an algorithm that uses more than locally optimal plans to find the best global plan Their algorithm is a Branch and Bound algorithm [RICH83] The assumption they make is that queries involve only equijoins and selections of the form $RA=cons$ This section presents a similar algorithm which is defined as a state space search algorithm (A* algorithm [RICH83]) with better average case performance than the one of [GRAN80] A model of queries with equalities only, makes our definition of $\Rightarrow$ redundant since it is subsumed by the definition of $\equiv$ We will also make here the same assumption to simplify the presentation of the algorithm At the end of this section we will discuss the extensions that can be done to include more general predicates in queries

As the original global query optimization problem was stated, we are 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 a set of plans (graphs) that can be used to process query i Good candidates for such plans are the locally optimal plans and the ones that use all common subexpressions among the given queries For example, for the queries Q_3 and Q_4 of section 3, in addition to the plans P_3 and P_4 presented there, we should also consider the plan

(P_{32}) retrieve into TEMP1 (EMP all,DEPT all)
where EMP dept = DEPT dept
retrieve (JOB all,TEMP1 all)
where JOB job = TEMP1 job

for query Q_3 because it shares the join EMP dept=DEPT dept 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, in this algorithm we consider optimizing a set of queries instead of a set of plans Considering more than one candidate plans per query has the desirable effect of detecting and using effectively all common subexpressions found among the queries

In order to present the A* algorithm, we need to define a state space $\mathbf{S}$, the way transitions are done between states and the costs of such 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 = 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 $next$ $\mathbf{S} \rightarrow ZZ$ with $next(s_1) = \min\{j \mid p_j=\text{NULL}\}$, if $\{j \mid p_j=\text{NULL}\} \neq \phi$

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 *cost* $tcost(t)$ 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 we see that the way transitions are defined, we will always replace the first NULL entry of a state vector, say at position i, with a plan for the corresponding query Q_i Finally, we define the initial and final states for the algorithm The state $s_0=<\text{NULL,NULL}, \ldots ,\text{NULL}>$ 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_i to s_F is minimal among all paths leading form s_0 to any final state This is exactly what global query optimization should do 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 We introduce here such a function h $\mathbf{S} \rightarrow ZZ$ in the following way let $s=<p_1,p_2, \ldots ,p_n>$, then

$$h(s) = \sum_{i=next(s)}^{n} \left(est\_cost(P_i) - \sum_{m \in P \cap P_i} n_m \ est\_cost(m) \right)$$

where

$$P_i = \min_j [est\_cost(P_{ij})]$$

$$P = \bigcup_{i=1}^{next(s)-1} P_i$$

m is a task found in $P \cap P_i$ with n_m being the number of times task m appears in the sets P_i, $next(s) \leq i \leq n$, and the function *est_cost* is defined on tasks and given by a procedure to be discussed below For a plan p, we assume that

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

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 For the similar algorithm presented in [GRAN80], such a function is given This function though does not take into account all interactions among the queries In [SELL85b] we describe a better function which is derived in a similar manner but is closer to the actual cost Since it is known that the better we estimate the cost, the faster the algorithm converges [RICH83], using our estimates will result to better performance The preprocessing algorithm is described briefly in the Appendix What we detect is a set S' of plans that are candidates for the optimal solution Then, based on the cost function *cost* defined for tasks, we define the following function *coalesced_cost* on tasks [GRAN80]

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

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

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

Based on that function, *est_cost* is defined as follows

$$est\_cost(P_{ij}) = \begin{cases} coalesced\_cost(P_{ij}) & \text{if } P_{ij} \in S' \\ Cost(P_{ij}) & \text{otherwise} \end{cases}$$

Once we have found the estimated cost function *est_cost*, the A* algorithm is applied

The final algorithm is the following

ALGORITHM HS

[1] Apply the preprocessing algorithm of the Appendix

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

[3] For all queries with no representative plan in S', find the originally cheapest plan and put it in the final solution

[4] For the rest of the queries run the A* Algorithm using a heuristic estimate *est_cost* to find the remaining entries in the solution

A complete example of the algorithm can be also found in the Appendix

The estimated cost of the final access plan $P = s_F$ will be,

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

where here CS represents the total number of subexpressions found in the n queries (not plans as it was the case in algorithm D)

Some comments are appropriate at this point First, the A* algorithm with the new estimator function *est_cost* will not take more steps than the original A* algorithm presented in [GRAN80] (which uses *coalesced_cost* as its estimator function) This is based on the fact that for every task m it is true that $est\_cost(m) \geq coalesced\_cost(m)$ Therefore with the help of a known theorem (see [GRAN80] for proof) our algorithm, will give a solution in at most the same number of steps as the algorithm of [GRAN80] Second, we must note that this algorithm 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 we fill the state vectors (i e in ascending query index) may not result to the best utilization of common subexpression results For example, if Q_1 has a more restrictive selection than Q_2, it would be better to consider executing Q_2 first To fix this problem we simply change the transitions to fill not the next available 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 h is defined similarly with the difference that we do not consider now only identical tasks but pairs of tasks t_i and t_j such that $t_i \Rightarrow t_j$ and $t_j \not\Rightarrow t_i$ as well The generalized algorithm is also presented in detail in [SELL86]

7 CONCLUSION

We presented a set of algorithms that can be used for multiple query processing These algorithms were presented as pieces 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 Using a representation for access plans rather than queries, we capture more general optimization environments than the ones that were used up to now for common subexpression isolation

Some of the algorithms that we 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 The last (heuristic search) algorithm, is a variation of the algorithm for optimizing a set of relational expressions initially 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 using EQUEL/C [RTI84] and the version of INGRES that is commercially available A random number of queries were drawn from a set of 8 queries and then executed

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

All algorithms b) through d) have performed better than algorithm a) even in the presence of fast access paths (on the relations or the indexes) Table 1 summarizes the results of our experiments by showing the decrease of both the I/O (number of disk accesses) and CPU costs that algorithms b), c) and d) achieve compared to the cost of running the queries without any interquery (global) optimization The same series of experiments was also run over the same set of queries with various primary or secondary access structures introduced on the relations involved This way we could check the viability of these algorithms even in the presence of fast access paths The results showed similar figures as the ones of Table 1 lowered by 5-10% depending on the size of the involved relations The major idea is that the overhead of a secondary structure may be higher than the overhead of accessing an unstructured intermediate result, which makes global query optimization reasonable under such environments as well The exact details of our experiments are further described in [SELL86]

As interesting future research directions we view the development of efficient algorithms for common subexpression identification and the extension to the algorithms presented to cover more general predicates Also the problem of applying global query optimization techniques in recursive query processing [IOAN86] is also a very interesting area of research Finally, rule-based systems provide a good example for multiple query processing Rule-goal trees [ULLM85] have many common paths because of

Algorithm	Performance Improvement
Better Serial	30%
Decomposition	30%
Heuristic	40%

Table 1

rules defined using other rules This means that the rule-goal tree is of great assistance in the course of finding common subexpressions and deriving efficient execution sequences

Acknowledgements I would like to thank my advisor Prof 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 I also thank my colleague Yannis Ioannidis and the anonymous referees, for their criticisms and suggestions that greatly improved the presentation of this paper

8 REFERENCES

[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*, Vol (1), 1983

[CHAK84] Chakravarthy, U S, Fishman, D H and Minker, J, *"Semantic Query Optimization in Expert Systems and Database Systems"*, Proceedings of the 1st International Conference on Expert Data Base Systems, Kiawah Isl, SC, October 1984

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

[FINK82] Finkelstein, S, *"Common Expression Analysis in Database Applications"*, Proceedings of 1982 ACM-SIGMOD International Conference on 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, 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, New York

[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"*, Tech Report UKSC 0060, IBM United Kingdom Scientific Centre, 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 Databases"*, PhD Thesis, University of California, Berkeley (in preparation)

[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

[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

[KUNG84] Kung, R et al, *"Heuristic Search in Data Base Systems"*, Proceedings of the 1st International Conference on Expert Data Base Systems, Kiawah Isl, SC, October 1984

[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

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

[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

[SELI79] Selinger, P et al, *"Access Path Selection in a Relational Data Base System"*, Proceedings of the 1979 ACM-SIGMOD International Conference on the Management of Data, Boston, MA, June 1979

[SELL85a] 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

[SELL85b] Sellis, T, *"Optimizing a Set of Relational Expressions"*, Manuscript, University of California, Berkeley, June 1985

[SELL86] Sellis, T, *"Optimization of Extended Relational Database Systems"*, PhD Thesis, University of California, Berkeley (in preparation)

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

[STON85] Stonebraker, M et al, *"The Design of POSTGRES"*, University of California, Berkeley, November 1985 (submitted for publication)

[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 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

We give here a brief description of our algorithm that derives the *est_cost* function

We will assume that we are given the n sets of plans $\mathbf{S}_1, \mathbf{S}_2, \ldots, \mathbf{S}_n$, with $\mathbf{S}_i = \{P_{i1}, P_{i2}, \ldots, P_{ik_i}\}$ We will also assume that we know the pairs of tasks $t_i \in P_{il}$ and $t_j \in P_{jm}$ that are $t_i \equiv t_j$ We define a directed graph $G(V,E)$ in the following way

- For each plan P_{ij} that has a task t_k identical to some task(s) of any plan(s) used for evaluating other than the i-th query, we introduce a vertex v_{ij}
- For each pair $t_i \in P_{kl}$, $t_j \in P_{pq}$ of identical tasks there is an edge connecting the two vertices $(v_{kl} \rightarrow v_{pq})$ if there is no other plan P_{pr} such that $t_s \in P_{pr}$ and $t_s \equiv t_i$

Given the above definition a unique graph can be built based on a set of plans Notice that not *all* plans are needed to build the graph Only those having identical tasks are used 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, we can keep only one edge $v_{ij} \rightarrow v_{kl}$ if there exists at least one edge between the two vertices v_{ij} and v_{kl} We loose no information that way because we do not really care how many identical tasks there are between the two plans, we only need to know if there is at least one

The goal of our algorithm is to find methods that are most probably not in the optimal solution The algorithm we use is a slightly modified Depth-First-Search (DFS) algorithm. The difference is that as we back up to the vertex v_{ij} from which we got to another vertex v_{kl} because of the edge $v_{ij} \rightarrow v_{kl}$, we store the identification (subscript) kl in some store associated with vertex v_{ij} Then, at the end of the algorithm, we delete from G all vertices that have two or more members $k'l'$ and kl in their associated store, such that $k' = k$ Along with the vertex, its edges (both out- and in-going) are also marked as OUT We continue this deletion process and delete vertices that have at least one out-going edge marked OUT The process stops when no more deletions are possible Let us call the final graph $G'(V',E')$ The set S' is the set of plans P_{ij} that have a corresponding vertex v_{ij} in G'

What we achieve with our preprocessing phase, is to reduce considerably the size of the search space for the A* algorithm. We need only consider plans for queries that have at least one plan still in the result graph G' The rest of the plans need not be considered since they do not interact with the others Therefore, it is sufficient to choose for the non present queries the cheapest plan and put it directly in the optimal solution The rest of the queries provide the remaining entries for the solution through the A* algorithm.

To give an example of the search and deletion process along with a run of the A* algorithm, we will use an example from [GRAN80] We are given two queries and 5 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

$$t_{11}^1 \equiv t_{21}^1 \quad , \quad t_{11}^2 \equiv t_{22}^2 \quad , \quad t_{12}^2 \equiv t_{23}^2 \quad ,$$

The graph of Figure 9 gives the graph G for the set of plans given After the DFS is performed, 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 too Finally, we see that no more vertices can be deleted The remaining graph is shown in Figure 10

Figure 10

From that graph we see that the final table for the estimated cost for the various tasks will be

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
Estim. Cost	40	30	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}
Estim Cost	75	45	55	50	40

Tracing the A* algorithm, we see that it visited 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 <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, we a total savings of 19% was achieved using the global optimization algorithm.

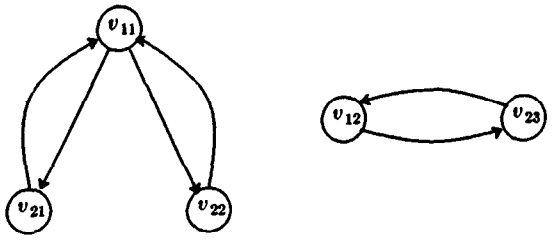

Figure 9

Comparing this with the function *coalesced_cost* used in [GRAN80] we would have

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
Coal Cost	20	15	5	35	10	10	5	10	10	30

and the coalesced costs for the plans are,

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

Tracing the A* algorithm again, 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 again <P_{12}, P_{23}> as the best solution But notice that with this set of estimators the algorithm exhaustively searches all possible paths in the state space