MULTIPLE QUERY OPTIMIZATION THROUGH STATE TRANSITION AND DECOMPOSITION

by

Wei Hong and Eugene Wong

Memorandum No. UCB/ERL M89/25

6 March 1989
(Revised May 31, 1989)

MULTIPLE QUERY OPTIMIZATION THROUGH STATE TRANSITION AND DECOMPOSITION

by

Wei Hong and Eugene Wong

Memorandum No. UCB/ERL M89/25

6 March 1989
(Revised May 31, 1989)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MULTIPLE QUERY OPTIMIZATION THROUGH STATE TRANSITION AND DECOMPOSITION

by

Wei Hong and Eugene Wong

Memorandum No. UCB/ERL M89/25

6 March 1989
(Revised May 31, 1989)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Multiple Query Optimization through State Transition and Decomposition

Wei Hong Eugene Wong
EECS Department, University of California
Berkeley, CA 94720

May 31, 1989

Abstract

There are two common sources of redundancy in multiple-query optimization algorithms: unnecessary ordering and unnecessary combination. In this paper, a new multiple-query optimization algorithm is proposed that eliminates these redundancies and achieves a near-linear complexity growth with respect to the number of queries to be jointly optimized. This algorithm makes use of two basic techniques: dynamic programming and decomposition. Dynamic programming is used to avoid repeating previous computation. Decomposition of multiple-query optimization into independent single-query optimizations linearizes the complexity growth. Both of these two techniques are based on a new state transition model for query processing. Sharing of operations between queries at different levels of abstraction is also achieved in this algorithm.

1 Introduction

Optimization of single queries has been one of the most successfully solved problems in database management systems. The solution is quite straightforward, that is, to do a systematic search over "all" possible plans[SELI79] [KOOI82]. No heuristic algorithms have proved to be adequate in practice. However, database applications are not always one-query-at-a-time. The following are some examples involving sets of queries to be optimized together.

- **Union Queries:** For example,

  ```
  (select name from emp where emp.salary < 15000)
  union
  (select name from emp where emp.name = student.name)
  ```

  A union query is usually processed as multiple queries by first evaluating all the subqueries and then computing the union. In some systems, even a single query with disjunctions in the predicate may result in multiple query processing, because the system takes a conjunctive query as the unit of processing.

- **Batched Queries:** In a distributed environment, we may have one master copy and several local copies of the same file and we periodically update the master copy with the updates to the local copies. This is a another example of multiple query processing in conventional databases.

- **Triggers:** In "active" databases, we have a set of triggers that are in the general form, *condition* → *action*, which means that as soon as the *condition* becomes true, the *action* is taken. To decide which actions to perform given a database state is also likely to be a multiple query processing problem.

- **Rules:** There is now general agreement that rules should be supported inside database systems. In a system supporting rules, such as POSTGRES[STON86], one simple query might be transformed into several queries applying different rules. For example, suppose we have the following rules,

  ```
  If employee's age is over 35, then his/her salary is 70% of
  his/her manager's.
  ```

If employee's age is below 35, then his/her salary is 50% of his/her manager's.

And we have the following query:
list the salary of employees in the toy department in SQL.

```
select salary
from emp
where emp.dept = 'toy'
```

This query can be converted into the following two queries to process.

```
select 0.5*mgr.salary
from emp, emp mgr
where emp.dept = 'toy'
and emp.age < 35
and mgr.name = emp.mgr
```

```
select 0.7*mgr.salary
from emp, emp mgr
where emp.dept = 'toy'
and emp.age ≥ 35
and mgr.name = emp.mgr
```

- **Procedural Data Type:** Procedural data type is an efficient way of representing complex objects in database systems. Querying a relation that contains procedural attributes may also convert a single query into several queries. The problem here is very similar to the case of rules.

Given multiple queries, current systems simply optimize them one by one and execute them separately. Let's call a plan to process a set of queries a *complete plan* and a plan to process a single query a *component plan.* A complete plan is just a combination of component plans. However, the optimal complete plan is not necessarily a combination of the optimal component plans. A combination of suboptimal component plans may do better if they can share some common costly operations (like joins) among the queries. This is the basic principle of multiple query optimization. An immediate idea of multiple-query optimization is to do a systematic search over all the combinations of all the possible component plans of each query. There are algorithms that simply do this as we shall survey in the next section. However, this is not practical, because the search space with multiple queries may grow intractable. Suppose the complexity (or size of search space) of each single-query optimization is C, a systematic-search multiple-query optimization of n queries will be of complexity C^n. Since each single-query optimization also does a systematic search, C is already big. A complexity

of C^n is simply unbearable. Fortunately, a complexity of C^n is not really necessary for multiple-query optimization because it contains redundancies. The redundancies, as we have discovered, can be classified as the following two cases:

- **Unnecessary Ordering:** If two operations in a plan are independent of each other, performing them in either order incurs the same cost. Thus they do not need to be considered as two different plans. For example, in a four-way join, $A \bowtie B \bowtie C \bowtie D$, $A \bowtie B$ and $C \bowtie D$ are completely independent. Doing them in either order will incur the same cost.

- **Unnecessary Combination:** If two queries have nothing in common, the optimal complete plan for processing them is simply the combination of the two optimal component plans. Thus considering any combination of their component plans is unnecessary. For example, the following two queries have nothing in common, thus any extra cost in addition to the sum of the two local optimization cost is undesirable.

```
select *                    select chairman
from emp                    from dept
where age < 50              where floor = 3
```

The purpose of this paper is to present a multiple-query optimization algorithm that eliminates these two forms of redundancies. Our basic approach is dynamic programming and decomposition. Dynamic programming technique avoids recomputing previous computation results over and over again. Decomposition, to be defined in the sequel, is a process that transforms a multiple-query optimization into a series of single query optimizations so as to reduce an exponential complexity growth to a near-linear complexity growth. Both of these two techniques are made possible by our state transition model of query processing, which is an extension to the model proposed in [LAFO86]. By parameterizing query processing by states, we achieve both clarity and efficiency. In multiple-query optimization, there is a tradeoff between the size of search space and the granularity of sharing. Our algorithm solves this problem gracefully by allowing the optimizer to work on different levels of abstraction, which is also based on our state transition model. At each state transition, the optimizer can "zoom" in on

lower level operations whenever there is an opportunity of sharing at the lower level.

This paper is organized as follows. Section 2 surveys relevant previous work and points out the problems in their algorithms. Section 3 defines a new state transition model for query processing that contains both logical and physical operations and specifies an optimization procedure for single queries that will be called in multiple-query optimization. Section 4 describes our multiple query optimization algorithm which decomposes multiple-query optimization into a series of single-query optimizations. Section 5 contains our conclusions.

2 Previous Work

Sharing is the basic principle of multiple-query optimization. This can be subdivided into two problems: first we want to identify sharable operations and second we need search strategies to decide which combination of component plans is optimal. Previous work has been done on both problems.

Sharable Operation Identification

A component plan consists of a sequence of *elementary operations*, which can be basic relational algebra operations or physical operations like sort and scan, depending on the level of abstraction we are working on. Note that these operations all have fixed input relations. For example, the selection, $\sigma_{emp.salary>60K}$ is an operation on relation *emp* only, not on any other intermediate relations derived from emp, e.g., it is different from the selection in $\sigma_{emp.salary>60K}(emp \bowtie dept)$, which is better represented as $\sigma_{(emp \bowtie dept).salary>60K}$.

The goal of multiple-query optimization is to share operations among queries. Sharable Operations can further be classified into operations that can be completely shared and operations that can only be partially shared. Suppose we have two operations p_1 and p_2. We define p_1 as being *equivalent* to p_2, denoted as $p_1 = p_2$ if the effect of p_1 is the same as the effect of p_2 under any database state. In this case, p_1 and p_2 can be completely shared. We define the *composition* of p_1 and p_2, denoted as $p_1 \circ p_2$ as the operation whose effect is the same as the effect of performing p_1 followed by p_2. If $p_1 \neq p_2$, but $p_2 = p_1 \circ p_{12}$ for some p_{12}, and p_{12} is of lower cost that p_2, than p_2, we say, p_2 *subsumes* p_1. We call p_{12} the *division* of p_2 by p_1. We denote it as p_2/p_1. Actually, equivalence is a special case of subsumption where $p_2/p_1 = I$. (I is the identity operation.) If p_2 subsumes p_1 then p_1 (part of p_2) can be shared. The most common form of subsumption is in the restriction and projection operations. For example,

$\sigma_{emp.salary>60K}$ subsumes $\sigma_{emp.salary>50K}$

$\Pi_{emp.name,emp.salary,emp.age}$ subsumes $\Pi_{emp.name,emp.age}$

There are also subsumptions between Sort and Group By and merge-join. For example, Group By subsumes Sort, Sort on multiple attributes can subsume Sort on a single attribute, Merge-join subsumes Sort.

To identify sharable operations is to find subsumptions between operations of different queries, since equivalence is a special case of subsumption. Most work on subsumption has been done on the relational algebra level, especially for restrictions and projections. [FINK82] only deals with equivalence. [CHAK86] identifies equivalence and subsumptions of logical operations by analyzing query graphs. [ROSE88] proposes a good algorithm for identifying subsumptions. The algorithm works on different levels of abstraction, both on logical level and physical level. We do not address the problem of identifying sharable operations in this paper. We assume that we have a procedure(e.g., the one proposed in [ROSE88]) for identifying common operations and concentrate on the search strategy problem.

Search strategies

There are two classes of search strategies: exhaustive search and heuristic search. [GRAN80] and [SELL86] propose exhaustive search algorithms. They basically search through all combinations of component plans with some complexity-control techniques like "branch-and-bound" and A^* algorithm. These algorithms all have the problem of redundancy in search space as we noted in the previous section. [CHAK86] proposes a heuristic search algorithm that generates directly a complete plan without searching. It has very low complexity, but is very likely to generate a suboptimal plan. [ROSE88] proposes AND-OR graph as the basic data structure for multiple query optimization, but it does not have any algorithm to search the AND-OR graph to find an optimal complete plan.

3 The State Transition Model

Problem Statement

The queries that we treat are of the following general SQL form,

```
select target-list
from relation-list
where qualification
[group by attribute-list]
[order by attribute-list]
```

The task of single-query optimization is to break a given query into sequences of elementary operations and find an optimal processing plan among them. In the context of multiple-query optimization, this is called *sub-optimization* as opposed to *complete optimization* which finds the optimal plan for processing a set of queries. A plan to process a single query is a *component plan* as opposed to that to process a set of queries, a *complete plan*. A complete plan is a combination of component plans. A component plan of a query is a sequence of operations that produces the result of the query. The operations that we have in a plan vary according to the level of abstraction being used. We have the following possible levels:

- **SPJ level.** In single query optimization, we have the rule that selections and projections are performed together with joins, i.e., they basically work as filters on the operands and/or results of joins. At this level, the operations in a component plan are SPJ's (selection-projection- join's). The local optimizer find the optimal sequence of SPJ's to process the query. Within each SPJ, where and how to perform selections and projections (e.g., index scan vs. sequential scan, pre-join vs. after-join, etc.) are decided by another level of suboptimization. The SPJ level is the most suitable level for single-query optimization.

- **Logical level.** On this level, the operations in the plans are basic relational algebra operations: selection, projection, join, etc.

- **Physical level.** On this level, the operations in the plans are major steps in the implementation of relational algebra operations: sequential scan, index scan, sort, nest-loop, merge-sort, etc.

The higher level operations are made up of sequences of lower level operations. In other words, any higher level is an abstraction of a lower level and any lower level is an implementation of a higher level. For single-query optimization, SPJ level is the best because it has the highest level of abstraction with the smallest search space, and the SPJ details are encapsulated in other independent suboptimizations. However, for multiple-query optimization, it is no longer the case, because it has the coarsest granularity of sharing. If we work at the SPJ level, the finest operations we can share is an SPJ. This would prohibit some sharable selections, or joins that are not in a common SPJ. Thus for multiple-query optimization, we may have to go to some lower levels, because even operations on physical level also have very significant cost. However, there is a tradeoff between the size of the search space and the granularity of sharing. At higher levels, the search space is small, but the granularity of sharing is also coarser, i.e., less opportunity of sharing. At lower levels, the search space is bigger, but the granularity of sharing is finer, i.e., greater opportunity for sharing. This is also a tradeoff between complexity and optimality. The best way to resolve this tradeoff is to allow the optimizer to work on mixed levels of abstractions. This is the strategy used in our algorithm. The way it works is that we normally search at the logical level but "zoom" in on physical level on detecting opportunities of sharing with finer granularity. All the existing algorithms either work at one specific level, e.g., [CHAK86] at logical level and [SELL86] at physical level, or work at a fixed but unspecified level, e.g., [ROSE88].

Multiple query optimization is built on top of single-query optimization. It imposes some special requirements on the single query optimizer. Not all single query optimizers meet these requirements. The basic requirements are listed below:

1. The optimizer must be able to generate suboptimal plans. This is because the optimal complete plan might be a combination of suboptimal component plans.

2. The optimizer must be able to compute and save partial costs. This is needed for decomposing a multiple-query optimization into a series of single query optimization.

3. We want a method that is able to work at mixed levels of abstraction.

Given these factors, a state-transition model seems to be the most appropriate for our purpose because in a state-transition model we are able to

save the information for all transition paths and store the partial processing costs in the states. We can easily "zoom" in on a lower level by expanding a transition into finer steps.

State Transition Model

Our state transition model is a modification of the one in [LAFO86] which was proposed for distributed query processing. The major differences are that we use **query graph**(as defined momentarily) instead of a collection of data to represent the states and allow operations from mixed levels of abstraction. Our state transition model also provides automatic elimination of the unnecessary-ordering redundancies.

To represent the states of query processing more easily, we introduce a graph representation called, *query graph*. A query graph consists of two kinds of nodes and edges connecting them. The nodes are relation nodes and operation nodes. Relation nodes represent relations, both base relations and relations derived from intermediate operations. Operation nodes represent relational operations. We depict relation nodes as rectangles and operation nodes as ovals. Edges connect operations with their operands.

Example 3.1

The query graph of the following relations and query is given in Figure 1.

```
cust(cno, name, age)
order(cno, date, item)

select *
from cust, order
where cust.age < 40 and
cust.cno = order.cno
order by order.date
```

If an operation is performed, we combine the operation node together with its operand node(s) into a new relation node. For example, after the join in the above query is performed, the resulting query graph becomes Figure 2, which represents the query that remains to be evaluated.

Now we can think of query processing as shrinking the query graph by combining nodes. At the final step, the graph will be reduced to one node, which is the query result.

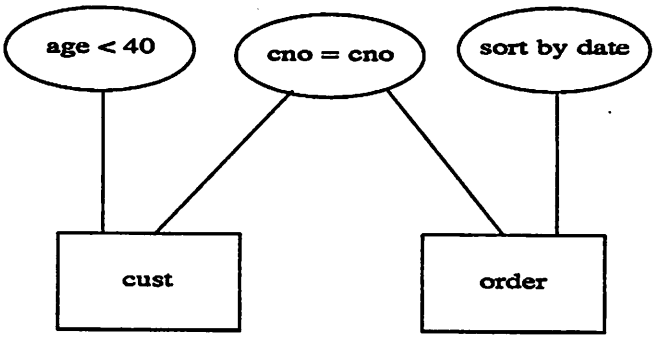

Figure 1: Initial Query Graph of Example 3.1

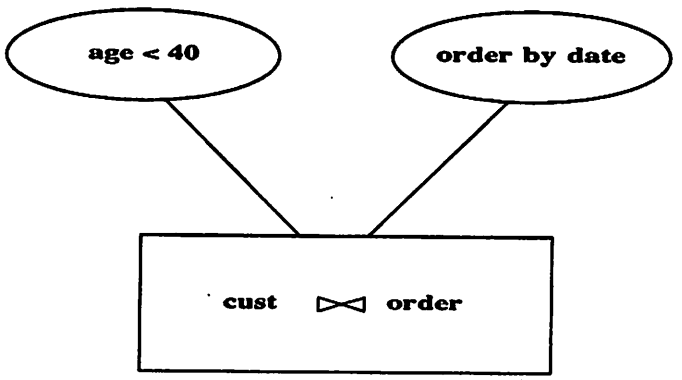

Figure 2: Resulting Query Graph of Example 3.1 after Join

A query graph clearly indicates which parts of the query remain to be done, what is already done, and what the possible operations are. This is why we choose query graph to represent the states of query processing.

Our state transition model is defined as follows.

For a single query Q, the state transition representation $ST(Q)$ is defined as

$$ST(Q) = (S, P, \Gamma, x_0, x_f)$$

where the different items in the collection are defined as follows:

S is the set of states, each state is represented by a query graph which defines the query remaining to be evaluated in the state.

P is the set of possible operations in the query processing.

Γ is the transition function: $S \times P \rightarrow S$. For $x \in S$ and $p \in P$,

$$\Gamma(x, p) = \text{the state result after applying } p \text{ on } x.$$

We can also write it as $x \circ p$. Not all operations in P are applicable to a state x. We define the set of operations which can be applied to x as $P(x)$.

x_0, x_f are initial and final states. x_0 is the state corresponding to the initial query graph, x_f is the state corresponding to the query graph with a single node.

In general, we define a *plan* for each state x as a sequence $p_1, p_2, \ldots, p_n$ such that $(\ldots((x \circ p_1) \circ p_2) \circ \ldots) \circ p_n = x_f$. A *component plan* of Q is a plan of state x_0.

For each state we introduce a cost function $C : S \rightarrow R$(Real Numbers).

For $x \in S$,

$$C(x) = min\{\sum_{i=1}^{k} cost(p_i) \mid x \circ p_1 \circ p_2 \ldots \circ p_k = x_f\}$$

where $cost(p_i)$ is the cost of operation p_i. Obviously, we have $C(x_f) = 0$. Now the optimization problem becomes one of computing $C(x_0)$.

The following dynamic programming formula can be used to compute $C(x)$.

$$C(x) = \min_{p \in P(x)} \{Cost(p) + C(x \circ p)\}$$

This formula contains the first kind of redundancy, i.e., different permutations of the same sequence of independent operations may get considered over and over again. To avoid this problem, we introduce the concept of *active* operations and *inactive* operations for each state. Initially, all the operations are active. After we consider one operation, we make it inactive so that it will not be considered again. The new formula can be written as follows.

$$C(x, P_{inactive}) = min(cost(p) + C(x \circ p, P_{inactive}), C(x, \{p\} \cup P_{inactive})) \quad (3.1)$$

where $x \in S, p \in P(x) - P_{inactive}$.

$P_{inactive}$ is the set of operations that are excluded from consideration. We use it to avoid unnecessary ordering of operations in plans, thus reducing the search space. The idea is that if an operation is not needed now, it will never be needed. This eliminates the first source of redundancies.

The following Theorem ensures the correctness of our formula.

Theorem 1 *The recurrence equation (3.1) yields the minimum cost for each state.*

Proof: Let $PL = \{p_1, p_2, \ldots, p_n\}$ be any plan of state x. If $p \in PL$, suppose $p = p_i$, let $PL' = \{p, p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n\}$. Since $p \in P(x), PL'$ is also a plan of x. And $Cost(PL) = Cost(PL') \geq Cost(p) + C(x \circ p, P_{inactive})$. If $p \notin PL, Cost(P) \geq C(x, \{p\} \cup P_{inactive})$. Thus, $Cost(PL) \geq C(x, P_{inactive})$, and the equation (3.1) is correct.

Q.E.D.

State Transition of Example 3.1

Figure 3 shows the state transition graph of the query in Example 3.1. Several points are worth noting in this example.

- At state x_2, the operation ($\sigma_{age<40}$ cust) is a possible operation, but it has been considered before, thus is inactive (marked by a cross). It is not considered again. Thus, orders of the independent selection and sort are not repeatedly considered.

- Figure 3 contains mixed levels of abstraction. We see that most of the transitions are due to logical operations, but some of the transitions (those inside the boxes) are due to physical operations such as

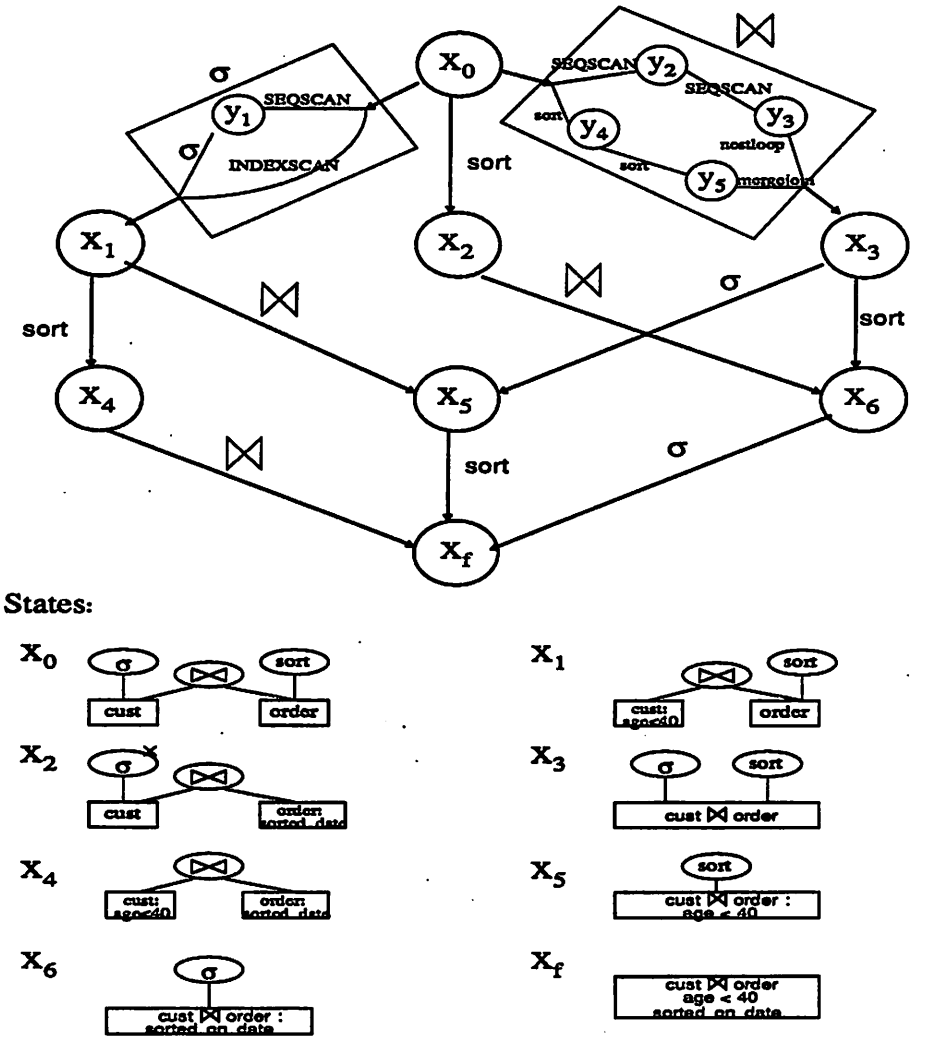

Figure 3: State Transition Graph of the Query in Example 3.1. *The query in Example 3.1 involves three operations to perform: select(σ) age < 40, join($\bowtie$) on cno and sort on date. The states marked as x_i result from logical operations and states marked as y_i result from physical operations. Physical level transitions are enclosed in boxes. Each box corresponds to a logical operation. We can "zoom" into a box to physical level transitions whenever necessary.*

sequential scan(SEQSCAN). These mixed levels of operations fit very well in our state transition model. A logical level transition is realized by several physical level transitions. If in multiple-query optimization we need to go to the physical level to explore finer granularity of sharing, we can simply expand a corresponding logical level transition into several physical level transitions. In this way, we can achieve a finer granularity of sharing without expanding the size of the entire graph excessively.

- We have to make more subtle differences between states resulted after the transitions of physical level operations. For example, state x_0 and state $y_1$contains the same collection of data, the only difference is that in y_1, the tuples of relation cust are buffer-resident instead of disk-resident.

4 Multiple Query Optimization

In this section we present our multiquery optimization algorithm. The basic motivation of our algorithm is to reduce the exponential complexity growth to as nearly linear as possible. Our strategy is to scrutinize the search space of multiquery optimization to eliminate all redundancies. We expand the search space from the union of search spaces of single-query optimizations only when it is absolutely necessary. The crux of our algorithm is the following two new ideas:

- **Decomposition.** Decomposition is our main approach to linearize the complexity growth. We decompose multiquery optimization into separate single-query optimizations as soon as we detect the fact that the queries remaining to be evaluated have nothing sharable.
- **Mixed Levels of Abstraction.** We do not restrict the algorithm to work at a specific level, or move from level to level. If we want to explore sharing at a lower level, we need only to refine the search space in a local area. In this way, we can encapsulate all the details that are irrelevant to the complete optimization in the suboptimizations so that the search space of complete optimization becomes as small as possible.

Both these techniques are based on the state transition model we proposed previously.

The state transition model for multiple-query optimization is a straightforward extension to the one for single-query optimization presented in the previous section.

For a set of queries, $\overline{Q} = \{Q_1, Q_2, \ldots, Q_n\}$, let $ST(Q_i) = (S_i, P_i, \Gamma_i, x_{i_0}, x_{i_f})$. The state transition representation $ST(\overline{Q})$ is defined as

$$ST(\overline{Q}) = (\overline{S}, \overline{P}, \overline{\Gamma}, \overline{x_0}, \overline{x_f})$$

where

$\overline{S} = S_1 \times S_2 \times \cdots \times S_n$

$\overline{P} = \cup_{i=1}^{n} P_i$

$\overline{\Gamma} : \overline{S} \times \overline{P} \rightarrow \overline{S}$, $(x_1, x_2, \ldots, x_n) \circ p = (x_1 \circ p, x_2 \circ p, \ldots, x_n \circ p)$. (If $p \notin P(x_i), x_i \circ p = x_i$).

$\overline{x_0} = (x_{1_0}, x_{2_0}, \ldots, x_{n_0})$

$\overline{x_f} = (x_{1_f}, x_{2_f}, \ldots, x_{n_f})$

A *complete plan* of a state $\overline{x}$ is a sequence of operations $p_1, p_2, \ldots, p_n$ such that $(\ldots((\overline{x_0} \circ p_1) \circ p_2) \circ \ldots) \circ p_n = \overline{x_f}$. We can define the cost of each state the same way as in single-query optimization, and we get the same recurrence equation,

$$C(\overline{x}, \overline{P}_{inactive}) = min(cost(p) + C(\overline{x} \circ p, \overline{P}_{inactive}), C(\overline{x}, \{p\} \cup \overline{P}_{inactive}))$$

where $\overline{x} \in \overline{S}, p \in P(\overline{x}) - \overline{P}_{inactive}$. $\overline{P}_{inactive}$ is a set of operations which are excluded from consideration.

If we follow this recurrence equation to do the global optimization, we search through the product space of all the local states. Thus the complexity is the product of the complexities of all local optimizations. This is an upper bound of the optimization complexity, but it is too big to be a useful upper bound. In the extreme case, the queries have nothing sharable, i.e., involving completely different set of relations, then the queries can be optimized separately and the complexity is only the sum of all the single-query optimization complexities. This is the lower bound. The goal of our new algorithm is to reduce the redundancy in search space such that the complexity decreases as close to the lower bound as possible. As we pointed out previously, there are basically two kinds of redundancies. One of them, unnecessary ordering is already taken care of by the recurrence relation itself. Now we need to deal with the unnecessary combinations, which is a more serious class of redundancy. Our basic strategy is decomposition. As soon as we find that there is nothing can be shared among the queries in the remaining steps, we decompose them into separate suboptimizations. Decomposition is based on the following equivalence relation.

Let CON be a binary relationship between single-query states. For $x_i \in S_i, x_j \in S_j$, x_i CON x_j if $P(x_i) \cap P(x_j) - \overline{P}_{inactive} \neq \phi$, or exist $p_1 \in P(x_i), p_2 \in P(x_j)$ and p_1, p_2 are both active such that p_2 subsumes p_1, or vice versa. Let CON* be the transitive closure of CON. CON* is an equivalence relation between single-query states. A multiple-query state $\overline{x}$ is a set of single-query states, therefore, CON* partitions a multiple-query state into equivalent classes (smaller multiple-query states). A multiple-query state is said to be *decomposable (partitionable)* if it can be partitioned by CON* into more than one equivalence classes. Obviously, queries in

different equivalence classes are completely independent of each other, thus can be optimized separately.

Decomposition Algorithm

Let $CP(\overline{x}) = \{p \in P(\overline{x}) \mid exist\ x_i \in S_i, x_j \in S_j, s.t.\ p \in P(x_i) \cap P(x_j), or\ exist\ p' \in P(\overline{x}), s.t.\ p'\ subsumes\ p\}$. $CP(\overline{x})$ is the set of sharable operations at state $\overline{x}$.

Our decomposition algorithm is described in pseudo-C as follows .

```
C(x̄, P̄_inactive)
x̄:  a set of queries
{
    if (partitionable(x̄))
    {
        partition x̄ into equivalence classes x̄_1, x̄_2, ..., x̄_k;
        return(Σ_{i=1}^{k} C(x̄_i));
    }
    else
    {
        pick p ∈ CP(x̄) − P̄_inactive;
        return(min(cost(p) + C(x̄ ∘ p, P̄_inactive), C(x̄, {p} ∪ P̄_inactive)));
    }
}
```

The basic ideas of our algorithm can be summarized as the following:

- We only consider sharable operations as state transitions at multiple query level. No sharable operations will be considered in separate single-query optimization.
- Each time a sharable operation is performed or forbidden, the possibility of partitioning of the set of query into equivalence classes also increases. At this time we check if the set of query has become decomposable. If it has, we immediately decompose them into independent subsets of queries and optimize them separately. In the end, it will run out of sharable operations (either performed or forbidden), and each

equivalence class will contain only a single query. Thus the multiple-query optimization problem is transformed into a set of single query optimization problems.

- The sharable operations can be either logical or physical. Therefore we are considering operations at mixed levels of abstraction, which is supported very well by our state transition model. Normally the state transitions are at the logical level. We go to the physical level only when a logical transition embodies sharable physical operations. logical transition.
- The condition when we "zoom' into physical level is very simple. If in the current state, there are two operations that involve one common input relation, we "zoom" the transitions corresponding to these two operations into physical level, because if the operations involve the same input relation, at least a SEQSCAN can be shared.

Example of Multiple Query Optimization

Consider the following 4 relations and 4 queries described in SQL. The corresponding initial query graphs are given in Figure 4.

Relations:

```
emp(eno, name, age, dept)
dept(name, chairman, floor)
cust(cno, name, age)
order(cno, date, item)
```

Q_1:

```
select *
from emp, dept
where emp.dept = dept.name
and dept.floor = 3
```

Q_2:

```
select *
from emp, dept
where emp.name = dept.chair
and dept.floor = 3
```

Q_3:

```
select *
from cust
where age < 30
order by name
```

Q_4:

```
select *
from cust, order
where cust.age < 40
and cust.cno = order.cno
order by cust.name
```

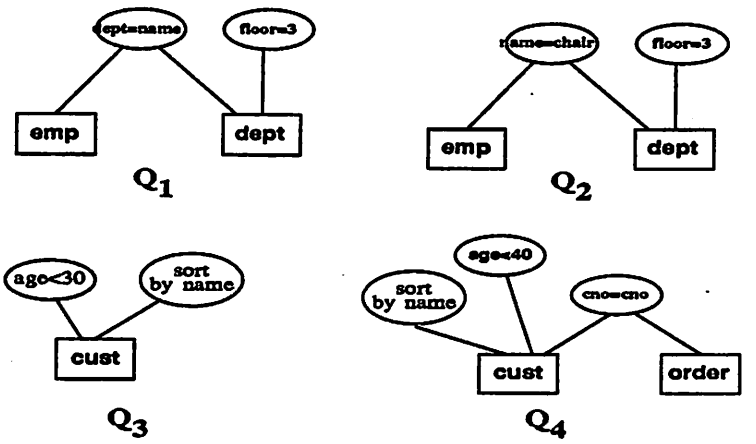

Figure 4: **Initial Query Graphs of Q1, Q2, Q3 and Q4**

We shall find an optimal processing plan for {Q1, Q2, Q3, Q4}. Obviously, at first this set of queries can be decomposed into two equivalence classes, {Q1, Q2} and {Q3, Q4}, which can be optimized independently as two separate groups. For {Q1, Q2}, the sharable operations are $\sigma_{floor=3}$, SEQSCAN on emp and nestloop join to form the product emp × dept. For Q3, Q4, the sharable operations are $\sigma_{age<40}$ and sort on name. The search process of each group is given in Figure 5 and Figure 6, in which we have three different types of nodes. The P-nodes are for "Partition". It partitions the set of query into equivalence classes and optimizes them separately. The B-nodes stand for "Branch". At a B-node, there are two branches coming out: one for the case that a sharable operation is performed and the other is for the case that the operation is deactivated (indicated by a '-'). The L-nodes stand for "Local optimization". At L-nodes, a multiple-query optimization has been decomposed into single-query optimizations. The optimization of Q1 and Q2 shows how we can search the state transition graph on mixed levels of abstraction. The optimization of Q3 and Q4 shows how we can take advantage of subsumption and share sort operations.

A processing plan for these 4 queries that maximizes sharing can be found by our algorithm as shown in Figure 7. However, a plan that maximizes the amount of sharing is not necessarily an optimal one. Sharing may incur cost.

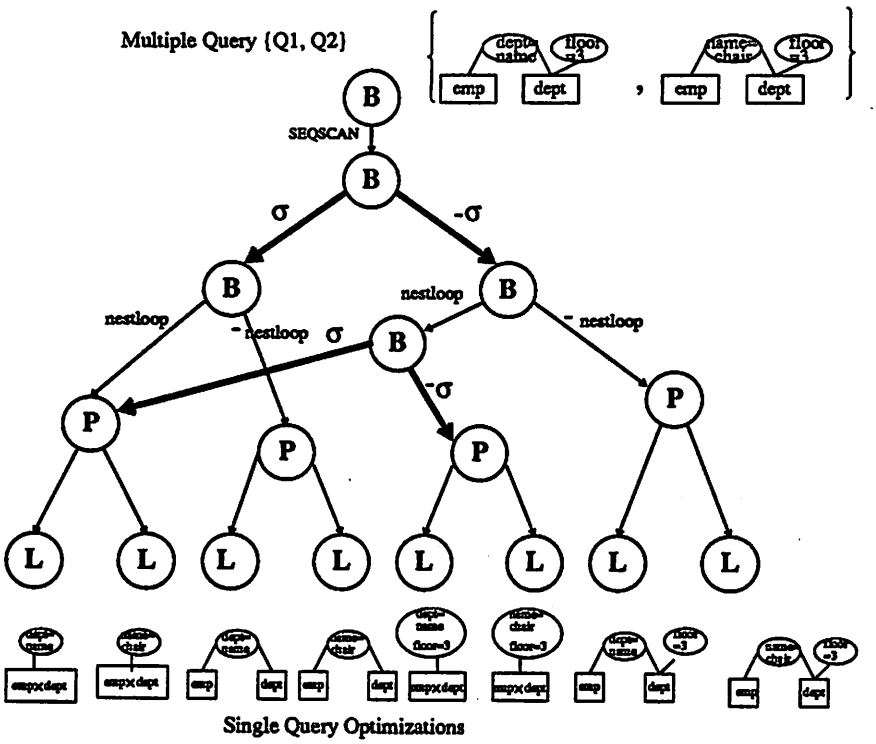

Figure 5: Multiple Query Optimization of $\{Q_1, Q_2\}$. *Bold arrows denote logical level transitions, others denote physical level transitions.*

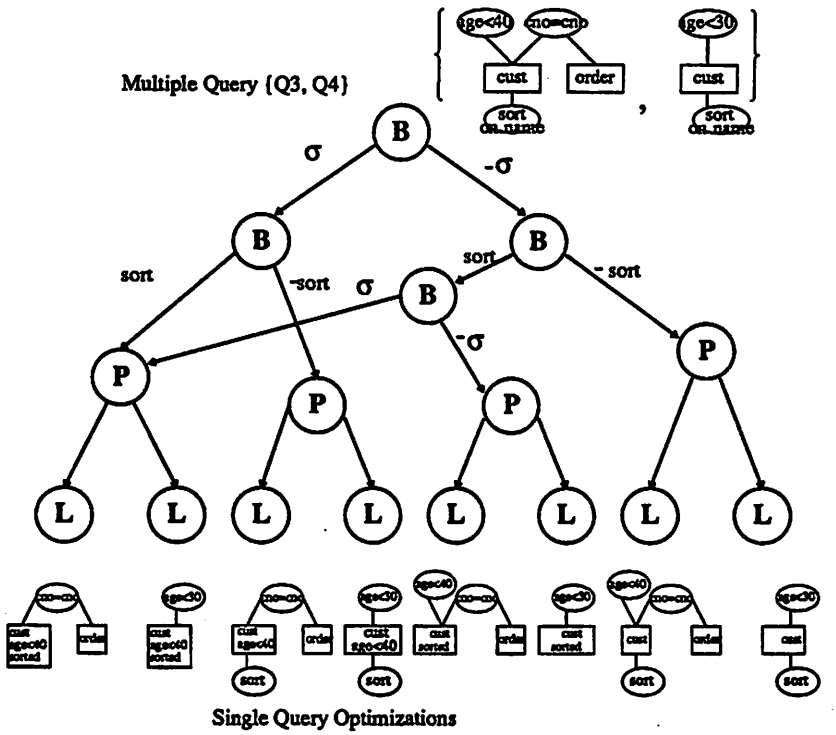

Figure 6: **Multiple Query Optimization of $\{Q_3, Q_4\}$**

For example, in Q1 and Q2 we have to use the inefficient nested-loop to do the join in order to share. Obviously this need not be optimal.

Complexity and Improvement

The complexity of our algorithm has the following two parts: (a) single-query optimization for each query separately and remember the cost of each state in the optimization, (b) for each sharable operation, deciding whether to share it or deactivate it. This can be expressed as the following:

$$O(\sum \textit{complexities of single query optimization} + 2^{\textit{number of sharable operations}}).$$

We believe that this estimate of complexity for our algorithm is a lower bound of the worst-case complexity for multiple-query optimization. However, we can add pruning to reduce the average complexity of our algorithm. "Branch and bound" seems to be most suitable for our algorithm. We can start with a bound of the sum of component plan costs with no sharing. Then, as we proceed, we reduce the bound whenever we find a better complete plan. We use the bound to prune branches of searches to reduce the complexity.

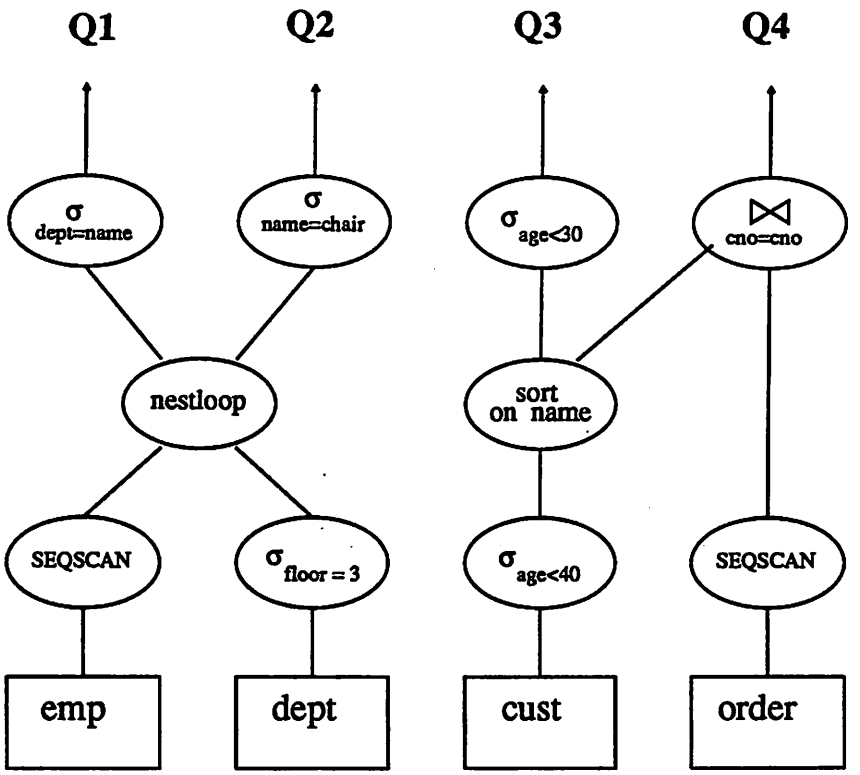

Figure 7: Complete Plan for $\{Q_1, Q_2, Q_3, Q_4\}$ with Maximum Sharing

5 Conclusions

Multiple query optimization is a complex combinatorial problem. To avoid intolerable combinatorial growth, we must eliminate redundancies that appear in almost all proposed algorithms, e.g., unnecessary ordering and unnecessary combination. The latter is more serious, because it increases the complexity exponentially. We believe that as a minimum requirement, given a set of independent queries, a good multiple query optimizer should spend no more than the sum of the time spent in optimizing single queries. In this paper, we propose a general algorithm that is free of these redundancies by using two techniques: dynamic programming and query-set decomposition. Both techniques are based on our state transition model of query processing, which allows transitions of mixed levels of abstraction. We believe the decomposition and "zooming" ideas are our major contributions. The "zooming" idea is tightly coupled to our state transition model, but decomposition is a general idea that can be applied to any existing algorithms to reduce complexity. We believe that the avoidance of unnecessary combinatorial growth achieved by these techniques is essential for making multiple-query optimization a practical reality.

References.

[CHAK86] Chakravarthy, U.S. and Minker, J., "Multiple Query Processing in Deductive Databases", University of Maryland, Technical Report TR-1554, College Park, MD, August 1985 Also in VLDB 86

[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

[GRAN80] Grant, J. and Minker, J., "Optimization in Deductive and Conventional Relational Database Systems," pp. 195-234 in Advances in *Database Theory 1*, H. Gallaire, J. Minker and J. M. Nicolas, eds. (Plenum Press, 1980)

[KOOI82] Kooi, R. and Frankforth, D., "Query Optimization in INGRES," IEEE Database Engineering, Sept. 1982.

[LAFO86] Lafortune, S. and Wong, E.,"A State Transition Model for Distributed Query Processing," ACM TODS, Vol. 11. No. 3, September 1986, pp 294 - 322.

[ROSE88] Rosenthal, A. and Chakravarthy, U.S., "Anatomy of a Modular Multiple Query Optimizer", Proc. of 14th VLDB conference, Los Angeles, California, 1988

[SELI79] Selinger, P.G., et al, "Access Path Selection in a Relational DBMS," Proc. of ACM SIGMOD, 1979

[SELL86] Sellis, T.K., "Global Query Optimization," Proc. of ACM SIGMOD, Washington, D.C.(May 1986).

[STON86] Stonebraker, M.R. and Rowe, L.A., "The Design of POSTGRES," Proc. of ACM SIGMOD, June 1986