A State Transition Model for Distributed Query Processing

STÉPHANE LAFORTUNE and EUGENE WONG
University of California, Berkeley

A state transition model for the optimization of query processing in a distributed database system is presented. The problem is parametrized by means of a state describing the amount of processing that has been performed at each site where the database is located. A state transition occurs each time a new join or semijoin is executed. Dynamic programming is used to compute recursively the costs of the states and the globally optimal solution, taking into account communication and local processing costs. The state transition model is general enough to account for the possibility of parallel processing among the various sites, as well as for redundancy in the database. The model also permits significant reductions of the necessary computations by taking advantage of simple additivity and site-uniformity properties of a cost model, and of clever strategies that improve on the basic dynamic programming algorithm.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; H.2.4 [**Database Management**]: Systems—*distributed systems*; *query processing*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*dynamic programming*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Distributed query processing, query optimization, semijoin, state transition model, tree queries

1. INTRODUCTION

We consider the problem of optimizing the processing of a query in a distributed database system. This problem has received a great deal of attention in recent research, and many algorithms for query optimization have been proposed and implemented. We refer the interested reader to [8, Chapter 6] and to the recent survey paper [25] for detailed reviews of the literature on this subject. References [1, 5, 9–13, 15, 18, 23, 24, 27] are of particular relevance to this paper. Generally

This research was sponsored by U.S. Army Research Office contract DAA29-82-K-0091, National Science Foundation grant ECS-8300463, and in the case of the first author, a scholarship from the Natural Sciences and Engineering Research Council of Canada.
Authors' addresses: S. Lafortune, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-1109; E. Wong, Department of Electrical Engineering and Computer Sciences and Electronics Research Laboratory, University of California, Berkeley, CA 94720.

speaking, most of the algorithms in these references fall into one of the following categories:

(i) those that give a local optimum or a "close to optimal" solution for general join queries, often by using heuristics based on the semijoin operation (cf. [1, 5, 13, 15, 23]);
(ii) those that give a global optimum for special classes of queries such as chain and tree queries that can be completely answered by semijoins (cf. [10, 11]); and
(iii) those that give a global optimum for general join queries, but for a class of strategies excluding the semijoin operation (cf. [18]).

Our objective is to formulate the distributed query processing problem within a precise state transition framework and then to find global optima among strategies based on joins and semijoins, using dynamic programming over the state space. In this sense, we believe that this paper generalizes the above references, in particular [5, 10, 11, 18, 23], which were our main inspiration.

The key element in this work is the introduction of a *state* to parametrize the evolution of the processing of a query in a distributed environment. Not only is this step crucial for modeling the dynamical nature of this problem, it is also necessary in order to use a dynamic programming algorithm to find the globally optimal solution. The cost of a state comprises all local processing and communication costs incurred in reaching the state.

Once the concept of state transition has been properly defined and the state space constructed, dynamic programming can be used to find the state containing the answer to the query that has the minimum cost and to find the optimal trajectory to that state (i.e., the optimal sequence of processing operations). We note that without a state transition model, the problem is not truly one of dynamic programming, and inefficient computations would probably result. An additional benefit of this framework is that clever strategies improving on the basic algorithm can be employed.

The concept of state can be thought of as a means for parametrizing the processing of a query in terms of joins and semijoins (which are a special case of a join followed by a projection). However, in contrast to many strategies that have been proposed (cf. [1, 5, 15, 23, 25]), we do not decompose the problem into a *reduction phase*, where only semijoins are used to reduce the relations, and an *assembly phase*, where all the joins are performed at a single site. Instead we consider a more general dynamical model allowing for arbitrary interleaving of join and semijoin operations, as well as for executing the joins in a distributed fashion (i.e., not all at the same site). The state trajectories in the state space will then include all the join orderings of [18], all the "correct nonredundant semijoin programs" of [10–11], and all the "semijoin reducer programs" of [5, 23].

Another feature of our model is that the definition of a state transition accounts for the possibility of parallel processing among the various sites where the database is located. We also allow a choice among multiple copies of a relation when the database incorporates redundancy. In the case where the sites are

uniform in terms of local processing and communication costs (as is assumed in almost all the literature), and where the answer can be located at any site (as in [5, 23]), we show how some states can be aggregated into equivalence classes, thus resulting in substantial savings for the computation of the optimal solution.

This paper is organized as follows. The problem is stated in Section 2. In Section 3, we present the state parametrization that we have formulated, and in Section 4 discuss the cost of one-step state transitions. The complete algorithm that we propose is given in Section 5. Section 6 is concerned with equivalence classes of states. A complete example is given in Section 7. Sections 8 and 9 generalize the results to the cases where semijoins are also allowed as state transitions, and where the database contains redundancy. Finally, we discuss our results in Section 10.

2. PROBLEM STATEMENT

Consider a distributed database system. By this we mean a database consisting of a finite set of *original relations* distributed among M sites, together with a collection of M autonomous processors communicating with each other via a general communication medium. The database may contain multiple copies of each original relation, but we assume that each copy is entirely located at a single site.

We are given:

(a) a query q_0 which references N distinct original relations not all located at the same site. We assume that the query can be described by means of the three relational algebra operations: projection, restriction (selection), and join. (We need not be more specific about the form of q_0 until Section 8.) As in [2], we call the *query graph* of q_0 the multigraph with N nodes, where each join clause in q_0 is indicated as a link between the corresponding nodes.

(b) an initial materialization (see [24]) of the N original relations of the database that are referenced by q_0. This consists of an M-component vector x_0, each component $x_0(i)$ containing those of the N original relations that are located at site i. In Sections 3 to 8, we assume that x_0 is irredundant (there is only one copy of each original relation), but this assumption is relaxed in Section 9.

The added dimension in distributed query processing, as compared to query processing in a centralized database system, is the necessity to transfer data when joining relations located at different sites. Thus the cost of processing a query includes both local processing and communication costs. In this paper we consider both categories and make no assumptions concerning their relative importance.

Finally, we assume that the site location of the answer to q_0 is irrelevant. But, as we indicate in Section 5, the algorithm that we propose also computes the optimal solution for all M possible locations of the answer. We use the terminology *site-uniformity assumption* to describe the situation where the processing costs are independent of the sites, and where the communication costs are the same between any two sites.

3. STATE PARAMETRIZATION OF THE PROBLEM

3.1 Notions of State and State Transition

We define a state x as follows: x is an M-component vector such that $x(i)$ contains the list of relations (including original relations and intermediate results) located at site i. The materialization x_0 is the initial state of the system. A final state is a state that contains the answer to q_0 at one of the M sites. We denote this answer by $q_0(x_0)$, and X_f denotes the set of all final states. In order to construct the set of reachable states, or state space, we need to specify the rules for state transitions. We do this by means of two definitions.

Definition. An *intermediate relation* derived from state $x \notin X_f$ is a new relation, that is, it is not already present in x, that can be obtained by (i) joining any two relations (original or intermediate) in state x, and then (ii) possibly making some projections and restrictions on this new result.

Therefore, all the relations in a state are either original relations or intermediate ones. Since any relation is always entirely located at a single site, all projection and restriction operations correspond to local processing (we adopt this terminology in the rest of the paper). Next, let $I = \{1, \ldots, M\}$.

Definition. We say that there exists a *one-step transition* from state x_1 to state x_2 if the following conditions are satisfied:

(i) $x_1 \notin X_f$;
(ii) $x_2 \neq x_1$;
(iii) for all $i \in I$, $x_2(i)$ is equal to $x_1(i)$, except for possibly one new intermediate relation derivable from x_1, and except for the deletion of some relations in $x_1(i)$ (deletion rules are specified later).

From the above definition, the new intermediate relation at site i is the result of a join between any two relations in x_1 (not necessarily located at site i), possibly followed by some local processing. In particular, the first definition allows for this new relation to be the semijoin of two relations in x, since a semijoin can be viewed as a join followed by a projection. (The operations need not be actually performed in that order; see Section 4.) We assume in this paper that all the semijoins are on the same attributes as the corresponding joins in the query graph of q_0.

Even though they are not natural relational algebra operations, semijoins are at the core of many distributed query processing algorithms (cf. [1, 5, 10, 11, 15, 23, 27]). For this reason, we have allowed for their explicit consideration in the state-transition framework. We emphasize that the above definitions permit arbitrary interleaving of *join state transitions* and *semijoin state transitions*, that is, semijoins need not be part of semijoin reducer programs only, as is the case in the above references.

Another feature of the model is that by allowing for one new intermediate relation at each of the M sites, and not only for one new intermediate relation from x_1 to x_2, we account for the possibility of parallel processing, in the sense

that we allow for simultaneously joining (and semijoining) distinct relations at different sites.

The modeling of the processing of a query by a state-transition model is influenced by the trade-off between *state* and *one-step state transition*. Finer definitions for state transitions, considering separately data movement and local processing, for example, result in a much bigger state space, and this may be computationally inefficient. On the other hand, coarser definitions, like allowing more than one join per site, may render the optimization of one-step transitions too complex and may cancel the advantages of using a state-transition model (dynamic programming is less advantageous when the number of steps is small). Moreover, in addition to being relatively simple to carry out, each suboptimal problem for the optimization of a one-step transition must be separable, meaning that it can be isolated from the rest of the problem, in order to be able to use dynamic programming. These considerations have led us to choose *join* as the unit step in state transitions, with the possibility of also allowing *semijoin* if a finer model is desired. If the cost function satisfies the site-uniformity assumption, the equivalence classes presented in Section 6 result in a coarser model.

We now divide the problem into two cases to simplify the presentation of our results. From this point on and until Section 8, we restrict ourselves to the case where the state transitions are joins only (i.e., semijoins are not allowed as one-step transitions). Observe that there is no restriction on how each such join is to be performed. Let A and B be two relations in x_1 and suppose that the transition from x_1 to x_2 is due to the operation $A \bowtie B$. Then $A \bowtie B$ could be the result of the *elementary semijoin program* $(A \ltimes B) \bowtie B$, but the intermediate step $A \ltimes B$ will not correspond to a state.

Our purpose is to exclude, for the moment, multistep semijoin programs which use sequences of semijoins on a relation to reduce it as much as possible in order to minimize the amount of data that has to be moved when the joins are actually performed. The general case including such sequences of semijoins for original and intermediate relations is more complex, and will be treated separately in Section 8.

The following deletion rule is adopted: when a relation is joined in a state transition, it is deleted from the new state. In the above example, this means that x_2 differs from x_1 by the addition of $A \bowtie B$ and the deletion of A and B.

A total of $N - 1$ joins (each possibly followed by some local processing) need to be performed to obtain $q_0(x_0)$, since we can assume, without loss of generality, that the query graph of q_0 is connected. (If disconnected, each connected part can be optimized separately.) Therefore, the above restrictions mean that a maximum of $N - 1$ state transitions are necessary to obtain a state in X_f from x_0. Observe that $card(X_f) = M$, namely one state for the answer at each of the M sites.[1]

In general, not all two original relations are joined in the query graph of q_0, and some transitions may correspond to joins that are in fact Cartesian products. Thus they may be expensive to perform. If one wishes to exclude these, as done in [18], for example, it is necessary to keep track of the new form of the query

[1] $card(Z)$ denotes the cardinality of set Z.

after each state transition to determine which joins are admissible. We do this by defining the complete state (x, q) where q is the updated form of q_0 when in state x, and of course $q(x) = q_0(x_0)$. In the following exposition of our solution method, for the sake of generality, we do not exclude joins that are Cartesian products. (We do so in the examples however.)

Example 3.1. Let q_0 be described by the query graph

$$A \text{ —— } B \text{ —— } C \text{ —— } D$$

which we simply write as $q_0 = A \bowtie B \bowtie C \bowtie D$, and let $x_0 = (A; B; C; D)$. Then, if $x_1 = (A \bowtie B; -; C; D)$, the new form of q_0 is $q_1 = (A \bowtie B) \bowtie C \bowtie D$. In other words, only two joins are admissible from x_1: $(A \bowtie B) \bowtie C$ or $C \bowtie D$.

3.2 Construction of the State Space

We use the notation $y \in T_j^+(x)$, if state y can be reached from state x in exactly j steps, for $j \geq 1$.[2] The fact that parallel processing is possible implies that for each state $x \neq x_0$, there exists integers $k(x)$ and $l(x)$ such that

$$x \in \bigcap_{j=k(x)}^{l(x)} T_j^+(x_0) \qquad 1 \leq k(x) \leq l(x) \leq N-1, \tag{3.1a}$$

$$x \notin T_j^+(x_0) \qquad \text{for} \quad 1 \leq j < k(x) \quad \text{and} \quad l(x) < j \leq N-1. \tag{3.1b}$$

$l(x)$ is easy to determine: add the number of joins that have been performed in the intermediate relations present in x. $k(x)$ depends on the amount of parallel processing that can be done in reaching x from x_0.

The state space, denoted X, is then

$$X := \{x : x \in T_j^+(x_0) \text{ for some integer } j,\ 1 \leq j \leq N-1\} \cup \{x_0\}. \tag{3.2}$$

Clearly, $X_f = T_{N-1}^+(x_0)$. $T^-(x)$ will denote the set of states that can reach x in a one-step transition:

$$T^-(x) := \{y \in X : x \in T^+(y)\}. \tag{3.3}$$

Our objective is to use dynamic programming to determine the minimum-cost trajectory from x_0 to any state in X_f. For this purpose, we have to divide the state space X into N disjoint subsets. (This is necessary to solve recursively the dynamic programming equation; see Section 5.) We subdivide X as follows:

$$X = \bigcup_{i=0}^{N-1} X(i), \tag{3.4a}$$

where

$$X(0) := \{x_0\} \tag{3.4b}$$

$$X(i) := \{x \in X : l(x) = i\}, \quad 1 \leq i \leq N-1, \tag{3.4c}$$

with $l(x)$ as defined in (3.1). The fact that l is a function on X implies that (3.4a) is true with the $X(i)$s mutually disjoint. Observe also that $X(N-1) = X_f$.

[2] When $j = 1$, it will be omitted as a subscript.

The motivation behind the above subdivision is to put a state in the indexed subset corresponding to the maximum number of steps in which this state can be reached from x_0. As a consequence, the following simple lemma is true.

LEMMA 3.1. *If* $x \in X(i)$, $0 < i \leq N - 1$, *and* $y \in T^-(x)$, *then* $y \in X(j)$ *with* $j < i$.

PROOF. Since $y \in T^-(x)$, $l(x) \geq l(y) + 1$, proving the result. □

4. ANALYSIS OF ONE-STEP STATE TRANSITIONS

4.1 Minimum Cost of One-Step Transitions

We now define the partial function $c: X \times X \rightarrow R^+ \cup \{0\}$ as follows. For $x \in X$ and $y \in T^+(x)$, $c(x, y)$ is defined to be the minimum cost of doing the one-step transition x to y. This transition involves doing one or more parallel joins between relations in x. (By parallel joins, we mean joins involving distinct relations with answers located at different sites.) This minimization problem has received much attention in the literature, and various methods have been proposed for joining two relations located at different sites. This point is discussed in the next section.

In this paper we do not study specifically how to compute $c(x, y)$, except for discussing what information is needed for its computation. We impose no assumptions on the function c, apart from requiring that it be nonnegative. Therefore, we allow for complete generality of the cost model.

Let $\gamma_S(x, y)$ denote a sequence of operations, comprising data movements and local processing, that performs the join and possibly subsequent local processing, in the transition from x to y. (In the case of parallel joins, assume γ_S is a vector whose components describe how each new intermediate relation in y is to be obtained from x.)

The important observation is that $c(x, y)$ is only a function of x, y and $\gamma_S(x, y)$, and does not depend on how the state x was reached from x_0. We refer to this fact as the *separation assumption.*

Remark 4.1. We use the terminology separation assumption, because, as mentioned in [18], the ordering of the tuples in the relations that are being joined can influence the cost of performing that join, depending on the way the data is accessed in the join method employed. We neglect such a dependency and assume throughout this paper that the separation assumption is valid. However, our model could also account for this further degree of refinement by including in the state information about the ordering of the tuples in each relation.

Letting Γ_S be the (finite) set of all admissible γ_S, we can write

$$c(x, y) = \min_{\gamma_S \in \Gamma_S} c(x, y; \gamma_S) \tag{4.1}$$

$$\gamma^*(x, y) := \operatorname*{argmin}_{\gamma_S \in \Gamma_S} c(x, y; \gamma_S), \tag{4.2}$$

where $c(x, y; \gamma_S)$ represents the total cost (including communication and local processing) to go from state x to state y by doing the operations described by $\gamma_S(x, y)$.

We now separate in γ_S the information concerning the relations that are joined from the specific site locations of these relations and the new intermediate one, in the following way:

$$\gamma_S(x, y) = g[\gamma(I(x, y)), s(x, y)] \tag{4.3a}$$

where, if we denote by R_1 and R_2 the two relations that are being joined, and by R_{1+2} the resulting new intermediate one,

$$I(x, y) := \{R_1; R_2; d; a\} \tag{4.3b}$$

with $d = 0$ if R_1 and R_2 are located at the same site in x, or $d = 1$ if not, and with $a = 1$, 2, or 3, according to whether R_{1+2} is located in y, at the site of R_1 in x, at the site of R_2 in x, or at some third site, respectively; and where

$$s(x, y) := \{\text{site of } R_1 \text{ in } x; \text{ site of } R_2 \text{ in } x; \text{ site of } R_{1+2} \text{ in } y\}. \tag{4.3c}$$

γ is to be seen as the restricted form of γ_S, depending only on $I(x, y)$, whereas the function g combines it with $s(x, y)$ to completely describe $\gamma_S(x, y)$.[3]

This notation is employed because under the site-uniformity assumption, a strategy for performing a join only depends on the information contained in $I(x, y)$, and not on the supplementary information in $s(x, y)$. We denote by Γ the set of all these strategies. Knowledge of $I(x, y)$ suffices to completely determine their costs. Typically, a strategy in Γ specifies which data is to be moved, for example, R_1 to the site of R_2, vice-versa, or both R_1 and R_2 to a third site, and how the join is to be performed, for example, merge join, nested-loop join, or by an elementary semijoin program. More details are given in the next section.

Therefore, we can write in this case

$$c(x, y) = \min_{\gamma \in \Gamma} c(x, y; \gamma) = \min_{\gamma \in \Gamma} c(I(x, y); \gamma), \tag{4.4}$$

the last equality emphasizing the information required for the computation of the cost. This result will be used in Section 6 to reduce the amount of computations under the site-uniformity assumption.

Finally, we mention that the evaluation of the function c requires information such as the size of the intermediate relations and the amount of processing time needed to perform some operation. In practice, these values are not known beforehand and estimates have to be found. This problem is not considered in this paper (see [25] for a review of some estimation algorithms). In any case, it is common to all distributed query processing algorithms, and we believe that the estimation task is no greater in our framework than in most other algorithms.

4.2 On Distributed Join Strategies

In this section we mention some strategies that can be included in the strategy space Γ of (4.4). At the outset, we point out that the state model presented in this paper is general enough to permit any distributed join strategy.

[3] In the case of parallel joins, think of all the above as vectors, each component associated with a different join.

Assume, as in [1, 5, 9–13, 15, 18, 20, 23–25, 27], that the site-uniformity assumption holds. $I(x, y) = \{R_1; R_2; d; a\}$ is given, and an appropriate set of strategies over which to carry the minimization in (4.4) must be determined. The decision on which strategies to include in Γ is a design problem that will be influenced by the specific cost model under consideration.

If $d = 0$, that is, if R_1 and R_2 are located at the same site, the work done in [6] suggests that one of the two join methods, nested-loop or merge-scan, will give good results. If also $a = 3$, the two relations may be moved and the join performed at the third site, or, instead, R_{1+2} may be moved to the third site.

In the case where $d = 1$, more options are available. In system R^* for example ([18, 20]), eight strategies γ are considered. These strategies are obtained by selecting interesting choices among all possible combinations of the following parameters: (i) join methods: nested-loop or merge-scan, and (ii) transfer strategy when moving relations: ship whole, ship whole and store, or fetch as needed. Also, the join is performed at the site specified by a, that is, the join site is that where R_{1+2} is located after the state transition. This requirement can be ignored to allow for more strategies.

When semijoin state transitions are not explicitly considered, as is assumed for the moment, elementary semijoin programs (Section 3.1) can also be included as strategies for the join state transition. These simple semijoin programs are often a useful tactic in query optimization, especially when communication costs are much more important than processing costs. (See [7] for recent simulation results on this issue.) Moreover, if multiprocessing at *each* site is available, suitable multiprocessor join algorithms can be included in Γ (see [22] for examples of such algorithms).

4.3 Additivity Properties of the Function c

Due to the possibility of parallel processing, two states that are connected by a one-step transition can also be connected by other j-step transitions, $j > 1$. In general, the total cost of each of these paths between the two states will not be the same. For example, if the two states differ by two new intermediate relations, it may be cheaper to do parallel processing and perform the two joins and required local processing in one step, if this is possible, than to do a 2-step transition. (This will be the case if c is the total *elapsed* time.) We are concerned with the following set of additivity properties for c.

Definition. Given a state space X, the function c defined in Section 4.1 is said to be

(i) additive if

$$c(x_1, x_n) = \sum_{i=1}^{n-1} c(x_i, x_{i+1}) \tag{4.5}$$

for all integers $n \leq N$ and for all $\{x_1, \ldots, x_n\}$ in X such that all the above terms are well defined. That is, we must have $x_n \in T^+(x_1) \cap T^+(x_{n-1})$ and $x_{i+1} \in T^+(x_i)$, $i = 1, \ldots, n-2$.

(ii) subadditive (superadditive) if

$$c(x_1, x_n) \leq (\geq) \sum_{i=1}^{n-1} c(x_i, x_{i+1}) \tag{4.6}$$

for all integers $n \leq N$ and for all $\{x_1, \ldots, x_n\}$ in X such that all the above terms are well defined.

A subadditive c means that parallel processing is always cost-advantageous. In this case, among all paths between any two states, one with only one step always has a nonsuperior cost to one with more than one step, as the above definition says.

It is reasonable to assume that c will either be additive or subadditive. For example, total processing time is additive, whereas total elapsed (or response) time is subadditive. Nevertheless, it may happen in some cases that c is neither. For the sake of generality we also take this case into account in the following sections. (Most of the work in the literature assumes an additive (cf. [5, 10, 11, 18]) or subadditive (cf. [1, 15]) cost function.)

5. ALGORITHM FOR THE COMPUTATION OF THE OPTIMAL SOLUTION

5.1 Dynamic Programming Equation

The algorithm that we propose is based on the separation assumption discussed in Section 4.1. The key fact about the solution to this problem is that it can be separated into two steps: (i) computation of the function c for all admissible pairs of states, and (ii) computation of the cost to reach each state by an application of dynamic programming.

The advantage of using dynamic programming is that we need not compute the costs of state trajectories, but only that of states. This results in substantial savings, since there is a maximum of $\sum_{i=0}^{N-1} card(X(i))$ states whose costs must be computed, whereas there can be as many as $\prod_{i=0}^{N-1} card(X(i))$ trajectories between x_0 and X_f. (Each trajectory corresponds to a distinct sequence of processing operations yielding $q_0(x_0)$ at some site.)

We now wish to describe step (ii) in detail. For this purpose, we must introduce new notation. For $x \in X$, we define $C(x)$ to be the minimum cost to go from state x_0 to state x, the number of steps being arbitrary. We also define $V(x)$ to be the minimum cost to go from state x to a state in X_f in an arbitrary number of steps. The boundary conditions are: $C(x_0) = 0$ and $V(x) = 0$ for all $x \in X_f$.

The objective is to determine

$$C(X_f) := \min_{x \in X_f} C(x) \tag{5.1}$$

along with the optimal state trajectory between x_0 and the state: $x_f^* := \text{argmin}_{x \in X_f} C(x)$. We also use the notation $C_j(\cdot)$ to denote the restriction of $C(\cdot)$ to $X(j)$.

The dynamic programming equation for this problem is

$$C(x) = \min_{y \in T^-(x)} [C(y) + c(y, x)]. \tag{5.2}$$

This is a consequence of the separation assumption. We want to specify an efficient recursive procedure for finding $C(x)$ for all states x. Such a procedure will depend on the additivity properties of c. In the case of additivity or superadditivity, the following lemma shows that we need only consider the set of trajectories with maximum number of steps between x_0 and x, since this set always contains an optimal solution.

LEMMA 5.1. *Suppose that c is additive or superadditive. Consider a state $x \in X(i)$. Then there is an i-step trajectory between x_0 and x that achieves $C(x)$.*

PROOF. Straightforward, using (4.5) and (4.6). □

When c is subadditive, that is, parallel processing is always advantageous, we can eliminate all the trajectories that contain a multistep path between two states whenever these two states can be connected by a one-step transition. In such cases the subadditivity property of c implies that the one-step path is more economical, and these trajectories will therefore never be optimal. However, this simplification in terms of trajectories is not immediately applicable to (5.2). To achieve it, we use $T_r^-(x)$, a subset of $T^-(x)$, constructed as follows.

Construction of $T_r^-(x)$

Step 1. Given an $x \in X(i)$, that is, $l(x) = i$, let j be the lowest integer such that

$$X(j) \cap T^-(x) \neq \emptyset.$$

Set $S(j) = T^-(x)$.

Step 2. If $j = i - 1$, then go to step 4. Otherwise, let $Z(j)$ be the set

$$Z(j) := X(j) \cap S(j) \tag{5.3}$$

and proceed to step 3.

Step 3. Determine the set $P(j)$, which is defined as follows:

$$P(j) := \{y \in S(j): y \in T+k(z) \text{ for some } 1 \leq k \leq i-j-1 \text{ and some } z \in Z(j) \text{ such that there is some optimal trajectory between } x_0 \text{ and } y \text{ that goes through this } z\}. \tag{5.4}$$

(Observe that the optimal trajectories to each such y are determined at the same time as $C(y)$, hence before the computation of $C(x)$ (from Lemma 3.1).)

Set $S(j+1) = S(j) - P(j)$, and then increment j and return to step 2.

Step 4. Set $T_r^-(x) = S(j)$.

For a motivation of the above construction procedure, refer to Figure 1. There, for the computation of $C(x)$, we can only remove y from $T^-(x)$ if we are sure that there exists a trajectory that reaches y by going throuch z and that achieves $C(y)$. If this is not the case, then

$$C(z) + c(z, y) > C(y) \tag{5.5a}$$

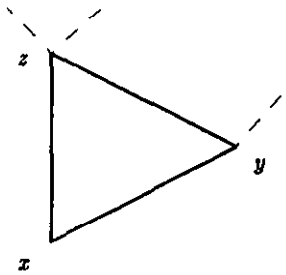

Figure 1

and we cannot conclude that

$$C(z) + c(z, x) < C(y) + c(y, x), \tag{5.5b}$$

even though

$$c(z, x) < c(z, y) + c(y, x). \tag{5.5c}$$

Consequently, y cannot be deleted from $T^-(x)$.

The following theorem gives a procedure for computing C over all the state space.

THEOREM 5.2. *Given the initial condition $C(x_0) = 0$, the function C can be recursively computed over all of X as follows. According to the properties of the function c, solve the following corresponding recursion for $i = 1, \ldots, N - 1$.*

Case 1. *c is additive or superadditive*:

$$C_i(x) = \min_{y \in T^-(x) \cap X(i-1)} [C_{i-1}(y) + c(y, x)]. \tag{5.6}$$

Case 2. *c is subadditive*:

$$C_i(x) = \min_{y \in T_r^-(x)} [C(y) + c(y, x)]. \tag{5.7}$$

Case 3. *c has none of the properties of cases* 1 *and* 2:

$$C_i(x) = \min_{y \in T^-(x)} [C(y) + c(y, x)]. \tag{5.8}$$

PROOF. First observe that the three recursions are well defined because, by Lemma 3.1, $T^-(x) \subset \cup_{j=0}^{i-1} X(j)$, and so all $C(y)$s on the right-hand sides have been computed before step i. By (3.4), $C(x)$ will be computed for all $x \in X$. What must be shown is that, in cases 1 and 2, the restriction on the domain of the optimizer y is of no consequence, that is, (5.6) or (5.7) return the same results as (5.2).

Case 1. c is additive or superadditive. By way of contradiction, suppose that there exists $z \in T^-(x) \cap X(j)$, with $j < i - 1$, such that

$$C_j(z) + c(z, x) < C_{i-1}(y) + c(y, x) \qquad \text{for all } y \in T^-(x) \cap X(i - 1). \tag{5.9}$$

The left-hand side of (5.9) describes a $(j + 1)$-step trajectory between x_0 and x whose cost is strictly smaller than all i-step ones, $i = l(x) > j + 1$. This is because

the right-hand side of (5.9) contains all i-step trajectories from x_0 to x, and only those ones. But, by Lemma 5.1, this means that c cannot be additive nor superadditive. We get the desired contradiction.

Case 2. c is subadditive. Again, suppose that there exist $w \in T^-(x) - T_r^-(x)$ such that

$$C(w) + c(w, x) < C(y) + c(y, x) \quad \text{for all} \quad y \in T_r^-(x). \tag{5.10}$$

But, from (5.4) in the construction of $T_r^-(x)$, there exists $z \in T_r^-(x)$ such that $w \in T_k^+(z)$, for some integer k, with z necessarily on an optimal trajectory between x_0 and w. That is, we can write

$$C(w) = C(z) + c(z, x_1) + \cdots + c(x_{n-1}, w) \tag{5.11a}$$

where we denoted by $\{z, x_1, \ldots, x_{n-1}, w\}$ this optimal path between z and w, or more simply:

$$C(w) = C(z) + c(z, w) \tag{5.11b}$$

if this path is only composed of one step. Combining (5.11a) and (5.11b) with (5.10) (setting $y = z$ there), we get, respectively,

$$c(z, x_1) + \cdots + c(x_{n-1}, w) + c(w, x) < c(z, x), \tag{5.12a}$$

$$c(z, w) + c(w, x) < c(z, w). \tag{5.12b}$$

But (5.12) means that there is a multistep path between z and x going through w, whose cost is strictly smaller than $c(z, x)$. This contradicts the subadditivity property (4.6) of c and demonstrates that no such w can exist. □

This theorem shows that when c possesses some additivity property, significant savings can be achieved for the computation of $C_i(x)$ by a restriction on the domain of the optimizer y. The minimization in (5.6) is over a considerably smaller set than the one in (5.2). (This is why we have chosen to put additive cs in case 1. They could also be considered as part of case 2.) In the subadditive case, if one does not wish to construct $T_r^-(x)$, then using (5.8) would of course yield the correct answer.

5.2 Statement of the Algorithm

We now have all the elements to state the algorithm that we propose.

Algorithm for Distributed Query Processing. Given a query q_0 referencing N original relations and an initial materialization x_0 for these relations in the distributed database, proceed as follows to determine the optimal sequence of processing operations for q_0, according to a given cost model.

Part I. Construct the state space X by constructing the sets $T_j^+(x_0)$, $j = 1, \ldots, N - 1$. At the same time, identify the sets $T^-(x)$ for all x in X. Subdivide X into disjoint subsets $X(i)$, $i = 0, \ldots, N - 1$ as described in (3.4).

Part II. Compute $c(x, y)$ for all $y \in X$ and for all $x \in T^-(y)$. $\gamma_S^*(x, y)$ of (4.2) gives the optimal sequence of operations for the state transition x to y. Determine if the function c possesses some additivity property.

Part III. Compute $C(x)$ for all $x \in X$ by using Theorem 5.2. For each x, let $y^*(x)$ be the argument (or the set of arguments) minimizing the appropriate form of the dynamic programming equation: (5.6), (5.7), or (5.8).

Part IV. The globally optimal cost is $C(X_f) = \min_{x \in X_f} C(x)$, achieved at x_f^*, say. The globally optimal trajectory(ies) is (are)

$$x_0, \ldots, y^*(y^*(x_f^*)), y^*(x_f^*), x_f^*, \tag{5.13}$$

that is, each one of them is constructed backwards from x_f^* until it reaches x_0.

The fact that the trajectory(ies) of part IV is (are) globally optimal is a consequence of the verification theorem for dynamic programming. If it is desired that the answer to q_0 be located at a specific site, then we eliminate the last minimization done in part IV: x_f^* is the final state in X_f that contains $q_0(x_0)$ at the given site. In addition, heuristics such as "defer to as late as possible joins requiring a Cartesian product" could be taken into account in the construction of the state space in part I.

Finally, the state model permits expression of the computational complexity of this problem algebraically in terms of the states. Because of the use of dynamic programming, the total number of additions and comparisons required in part III is at most quadratic in the cardinality of the state space. On the other hand, in the worst case where $N = M$ and where no heuristics are used to prune the state space, we have not been able to obtain a polynomial bound on the total number of states, and this number is probably exponential in N. (Observe that the equivalence class technique of Section 6 can significantly reduce that number.)

5.3 A "Best-First" Strategy

The algorithm of the preceding section gives a systematic way of computing $C(X_f)$. It is however possible to compute the optimal solution without necessarily having to compute all $c(\cdot\,, \cdot)$ pairs and all $C(\cdot)$. For example, suppose that we know an upper bound for the optimal cost, that is, suppose that we have determined that $C(X_f) \leq R$. Then, if we compute a $C(x) > R$ for some state x, we know that this state will never be on an optimal trajectory and need not be considered in the remaining calculations. This will result in smaller $T^-(\cdot)$ sets for all states in $T^+(x)$. In particular, the one-step optimal transitions $c(x, \cdot)$ need not be computed.

We now wish to propose a "best-first" strategy which takes advantage of this fact to improve on the efficiency of the basic dynamic programming algorithm of the last section. More precisely, we want to replace parts II and III of that algorithm by a better procedure. For this purpose, we need new terminology. We say that a state z is an *ancestor* of state x if there exists k, $1 \leq k \leq N - 1$, such that $x \in T_k^+(z)$. The set of all the ancestors of x is denoted by $Anc(x)$. Observe that our definition of a one-step transition implies that each state in X_f has exactly $X(N - 2)$ as the set of its ancestors. This observation justifies step 4 in the following procedure.

"Best-First" Strategy for Parts II and III of the algorithm. Given a state space X and $X_f \subset X$, we want to determine $C(X_f)$ and the optimal trajectory to the optimal state in X_f.

Step 1. Determine a good upper bound R_f such that $C(X_f) \leq R_f$, and denote the trajectory that achieves this value R_f by $Traj(1)$. $Traj(1) = \{x_0, \ldots, x_{N-2}(1), x_f\}$, where $x_{N-2}(1) := Traj(1) \cap X(N-2)$, and where $x_f \in X_f$ is chosen together with R_f and $Traj(1)$, or is the state containing $q_0(x_0)$ at the desired site, if a given site is specified for the location of the answer. Set $i = 1$ and $X^r = X$.

Step 2. Set $X_i^r = (Anc(x_{N-2}(i)) \cap X^r) \cup \{x_{N-2}(i)\}$. Use Theorem 5.2 to compute $C(x_{N-2}(i))$, considering X_i^r as the state space, and computing each $c(\cdot, \cdot)$ only when needed. If a state has cost greater than R_f, then it can be deleted whenever it appears in a subsequent $T^-(\cdot)$ set. Update the current bound R_f if there exists $x_f \in X_f$ such that $C(x_{N-2}(i)) + c(x_{N-2}(i), x_f) < R_f$, or if this is true for the specified x_f.

Step 3. Delete all unnecessary states from the current state space:

$$X^r = X^r - \{x: C(x) > R_f\}.$$

Step 4. If not all the remaining states in $X^r(N-2)$ have been used as an $x_{N-2}(j)$ for some $j \leq i$, set $i = i + 1$ and choose a new $x_{N-2}(i)$, under the constraint that the selected state has the smallest number of ancestors whose costs have not yet been computed, and then return to step 2. Otherwise, compute $C(X_f)$ and determine the optimal trajectory(ies).

This strategy can be used recursively (i.e., step 2 can be carried out by invoking the same best-first procedure with $x_{N-2}(i)$ in place of X_f, and with X_i^r in place of X. However, in step 3, X^r and R_f must always remain those corresponding to the original application of the procedure. This is due to the following observation. Let the upper bound for the first subproblem be denoted by $R_{N-2}(i)$. Then states with cost greater than $R_{N-2}(i)$ can temporarily be deleted from X_i^r for the purposes of the calculation of $C(x_{N-2}(i))$, but they should not be deleted from X^r unless their cost also exceeds R_f. This is because even though such states never lie on an optimal trajectory from x_0 to $x_{N-2}(i)$, they may lie on one from x_0 to X_f.

The determination of a good first upper bound in step 1 is normally based on heuristics. For example, it could be the result of another (fast) suboptimal algorithm, or it could correspond to any "initially feasible solution" (as defined in [24]). (Observe that it is required that the sequence of processing operations corresponding to this upper bound be a trajectory in the state space.) Clearly, the smaller this upper bound is, the higher the savings yielded by the best-first strategy are. Finally, we mention that the choice of the new $x_{N-2}(i)$ in step 4 could be made on different considerations than simply the number of ancestors whose costs remain to be computed.

(The A^* algorithm (see, e.g., [19]) has been suggested as a way of finding the optimal solution without necessarily having to compute the costs of all the states (or even having to construct them). However, an important consideration is that a conservative estimate of the cost-to-go $V(x)$ must always be available to guarantee that this algorithm will generate an optimal solution.)

6. EQUIVALENCE CLASSES OF STATES

We assume throughout this section that the site-uniformity assumption holds and that the site location of the answer is irrelevant. In this case, it is possible to aggregate states into equivalence classes and considerably reduce the computations required to optimally solve the problem.

Definition. Two states x_1 and x_2 are said to be *equivalent*, denoted $x_1 \approx x_2$, if the two following conditions are satisfied.

(i) For all $i \in I$ such that $x_1(i)$ or $x_2(i)$ contains original relations, $x_1(i) = x_2(i)$.
(ii) Letting all other sites be denoted by the set of indices $J \subset I$ (in other words, for all $j \in J$, neither $x_1(j)$ nor $x_2(j)$ contain original relations), $x_1(J)$ is equal to $x_2(J)$ up to a permutation of its components.[4]

LEMMA 6.1. *If* $x_1 \approx x_2$, *then*

(i) $T^+(x_1) \approx T^+(x_2)$, *where* $\approx$ *for sets means that each element on the left has a corresponding element on the right that is equivalent to it;*
(ii) $V(x_1) = V(x_2)$;
(iii) $l(x_1) = l(x_2)$ *and* $k(x_1) = k(x_2)$.

PROOF. (i) x_1 and x_2 differ by a permutation of components containing only intermediate relations. Do the same permutation on each state in $T^+(x_1)$. Since the characterization of a one-step transition is preserved under site permutations, the resulting set is $T^+(x_2)$. (ii) Follows from the definition of V in Section 5.1 and from the site-uniformity assumption. (iii) Immediate, since x_1 and x_2 contain the same relations. □

The proper interpretation to (ii) above is that the site permutation in going from x_1 to x_2 can be propagated along an optimal trajectory from x_1 to X_f to yield an optimal trajectory from x_2 to X_f. Observe, however, that in general $C(x_1) \neq C(x_2)$ even if $x_1 \approx x_2$.

We define an equivalence class of states, denoted $\mathbf{x}$, to be a set of states that are mutually equivalent. From now on we regard the state space X as the collection of all equivalence classes, each containing at least one state, and all of them being necessarily mutually disjoint. We wish to work with these equivalence classes directly. For this purpose, we define

$$T^-(\mathbf{x}) := \bigcup_{x \in \mathbf{x}} T^-(x), \tag{6.1}$$

and we say that $\mathbf{x} \in T^+(\mathbf{y})$ iff $\mathbf{y} \in T^-(\mathbf{x})$.[5] Because of Lemma 6.1 (i), this definition is sufficient for $T^+(\cdot)$, that is, we need not do as in (6.1). In particular, the following result is true.

COROLLARY 6.2. *Let* $\mathbf{y} \in X$, *and consider any* $y \in \mathbf{y}$. *Then, for each* $\mathbf{x} \in T^+(\mathbf{y})$, *there exists* $x \in \mathbf{x}$ *such that* $x \in T^+(y)$.

The purpose of the above is to extend the domain of definition of $c(\cdot, x)$ from $T^-(x)$ to $T^-(\mathbf{x})$. Whenever a state $y \in T^-(\mathbf{x}) - T^-(x)$, we set $c(y, x) := \infty$. This has two consequences. First, it implies that

$$\min_{y \in T^-(\mathbf{x})} [C(y) + c(y, x)] = \min_{y \in T^-(x)} [C(y) + c(y, x)], \tag{6.2}$$

[4] $x(J)$ denotes x restricted to its components in the index set J.
[5] In the following, we regard $T^-(\mathbf{y})$ as a collection of equivalence classes.

when $x \in \mathbf{x}$. Second, it makes the following definition consistent. For $\mathbf{y} \in X$ and $\mathbf{x} \in T^{-}(\mathbf{y})$, we define

$$c(\mathbf{x}, \mathbf{y}) := \min_{x \in \mathbf{x}, y \in \mathbf{y}} c(x, y). \tag{6.3}$$

Now, letting

$$C(\mathbf{x}) := \min_{x \in \mathbf{x}} C(x), \tag{6.4}$$

we can prove the following lemma.

LEMMA 6.3. $C(\mathbf{x}) = \min_{\mathbf{y} \in T^{-}(\mathbf{x})} [C(\mathbf{y}) + c(\mathbf{y}, \mathbf{x})]$.

PROOF. From (6.4), (5.2), and (6.2), we have that

$$C(\mathbf{x}) = \min_{x \in \mathbf{x}} \{ \min_{y \in T^{-}(\mathbf{x})} [C(y) + c(y, x)] \}. \tag{6.5}$$

But the two minimizations can be interchanged in (6.5), yielding successively

$$C(\mathbf{x}) = \min_{y \in T^{-}(\mathbf{x})} \min_{x \in \mathbf{x}} [C(y) + c(y, x)] \tag{6.6}$$

$$C(\mathbf{x}) = \min_{\mathbf{y} \in T^{-}(\mathbf{x})} \min_{y \in \mathbf{y}} \min_{x \in \mathbf{x}} [C(y) + c(y, x)] \tag{6.7}$$

$$C(\mathbf{x}) = \min_{\mathbf{y} \in T^{-}(\mathbf{x})} [C(\mathbf{y}) + c(\mathbf{y}, \mathbf{x})], \tag{6.8}$$

where (6.7) is obtained by breaking the first minimization into two steps (define the ys by partitioning $T^{-}(\mathbf{x})$ into disjoint equivalence classes), and where (6.8) is obtained by bringing the last two minimizations inside the brackets and by using definitions (6.3) and (6.4). □

This lemma shows that $C(\mathbf{x})$ does not have to be computed from its definition (6.4), but instead can be computed recursively by means of (6.8) and (6.1), starting from the initial condition $C(\mathbf{x}_0) = 0$ (with $\mathbf{x}_0 := \{x_0\}$). In other words, once c has been computed for all admissible pairs of equivalence classes, we only have to consider these equivalence classes, not the individual states they are composed of, for the computation of C. The desired answer is $C(X_f)$, since X_f is itself an equivalence class.

With (6.8) now established, it is clear that the generalization of Theorem 5.2 to equivalence classes is also true. It suffices to replace states by equivalence classes everywhere in the statement of that theorem. (The proof makes use of Lemmas 6.1 (iii) and 6.3, and of the following observation: the definitions of additivity and sub(super)additivity in Section 4.3 guarantee that (4.5) and (4.6) (and hence Lemma 5.1 as well) still hold when individual states are replaced by equivalence classes in these equations.)

Observe that so far we have not invoked the site-uniformity assumption. Solving (6.8) recursively yields $C(X_f)$ and one or more optimal trajectories going through equivalence classes and represented by

$$\mathbf{x}_0, \ldots, \mathbf{y}^*(\mathbf{y}^*(X_f)), \mathbf{y}^*(X_f), X_f, \tag{6.9}$$

where $\mathbf{y}^*(\mathbf{x})$ denotes the argument minimizing the given dynamic programming equation for $C(\mathbf{x})$. However, the definition of $c(\mathbf{x}, \mathbf{y})$ in (6.3) shows that it will

depend on specific states inside $\mathbf{x}$ and $\mathbf{y}$. Consequently, we must show that we can match the various paths between the equivalence classes in (6.9) to produce a continuous optimal state trajectory. For this, we must invoke the site-uniformity assumption.

Let x^* and y^* be two arguments minimizing (6.3). Since the site-uniformity assumption holds, we can make use of (4.4). Therefore, the minimum cost of the transition between x^* and y^* depends only on $I(x^*, y^*)$, and not on $s(x^*, y^*)$, and is achieved by the strategy $\gamma^* \in \Gamma$, say. Suppose now that the previous path in the optimal trajectory reaches state $x' \in \mathbf{x}$, but that $x' \neq x^*$. We know from Lemma 6.1 (ii) that these two states have the same minimum cost-to-go. Nevertheless, we want to be more precise concerning the continuity of the trajectory inside the equivalence class $\mathbf{x}$. The following lemma shows how to connect the two portions.

LEMMA 6.4. *Let x^*, y^*, and γ^* be arguments minimizing* (6.3) *and* (4.4). *Then, given $x' \approx x^*$, there exists $y' \approx y^*$ such that*

$$c(x', y') = c(x^*, y^*), \tag{6.10}$$

$$c(x', y') = c(x', y'; \gamma^*). \tag{6.11}$$

PROOF. Recall the definition of $I(x, y)$ in (4.3b). Clearly, from the definition of equivalence, the elements R_1, R_2, d depend only on the class $\mathbf{x}$. We can perform on y^* the same permutation that transforms x^* into x' to get y', with $y' \in T^+(x')$, since one-step transitions are preserved under site permutations. But, clearly, $I(x', y') = I(x^*, y^*)$, that is, by propagating the permutation we obtain in $I(x', y')$ the same a as in $I(x^*, y^*)$. (6.10) and (6.11) are then immediate from (4.4) □

Observe that this lemma states that not only the pair (x', y') has the same minimum transition cost as (x^*, y^*), but, moreover, that this minimum is achieved by the same strategy γ^*.

A consequence of this lemma is that the first time an optimal trajectory enters an $\mathbf{x}$ with $card(\mathbf{x}) > 1$ from some state z, $c(z, \mathbf{x})$ determines a specific state $x(z) \in \mathbf{x}$. Then, by way of the propagation of permutations of the above proof, specific states in all the subsequent equivalence classes in the trajectory are being determined. The resulting complete state trajectory achieves $C(X_f)$. This application of Lemma 6.4 is the only addition to the algorithm of Section 5.2 in the site-uniformity case considered in this section.

7. AN EXAMPLE

Let q_0 be described by the query graph of Figure 2, where each link represents a join.

Let $x_0 = (P, C; I; E)$, that is, the original relations P and C are located at site 1, I at site 2, and C at site 3. We want to find the optimal sequence of operations to obtain $q_0(x_0)$ from x_0, assuming the sizes of the original and intermediate relations are as in Figure 3, and for the following simple site-uniform cost model:

$c(x_1, x_2; \gamma)$ = total size of data moved between sites in the transition $\gamma(x_1, x_2)$.

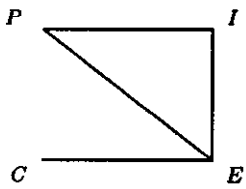

Fig. 2. Query graph.

RELATION	SIZE
P	1000
C	50
I	100
E	500
$P \bowtie I$	100
$P \bowtie E$	500
$I \bowtie E$	30
$C \bowtie E$	50
$P \bowtie I \bowtie E$	30
$C \bowtie E \bowtie I$	10
$P \bowtie E \bowtie C$	50
$P \bowtie I \bowtie E \bowtie C$	10

Fig. 3. Size of relations.

(This cost model does not consider local processing costs.) We solve this problem using equivalence classes, considering the following strategies in Γ:

(i) $d = 0$ and $a = 3$: move R_1 and R_2 to $s(R_{1+2})$, or move R_{1+2} to $s(R_{1+2})$;[6]
(ii) $d = 1$ and $a = 1$: move R_2 to $s(R_1)$, or move R_1 to $s(R_2)$ and then R_{1+2} to $s(R_1)$;
(iii) $d = 1$ and $a = 3$: move R_1 to $s(R_2)$ and then R_{1+2} to the third site, or move R_2 to $s(R_1)$ and then R_{1+2} to the third site, or move R_1 and R_2 to the third site.

The complete state space is give in Figure 4 (only one state per equivalence class is indicated), and the diagram of all state trajectories in Figure 5. From the data in Figure 3 and the above list of admissible strategies γ for a one-step state transition, it is straightforward to obtain from (4.4) the values of c labelling the one-step paths in Figure 5. Then, by an application of the dynamic programming equation (6.8) (or (5.6) with equivalence classes, since the above c is clearly additive), we obtain the $C(\mathbf{x})$ listed in Figure 4. We conclude that the optimal cost is 110, and that it is achieved by the four following trajectories, all giving the answer at site 1:

$$x_0 \to x_{10} \to x_{16} \to (P \bowtie I \bowtie E \bowtie C;\ -;\ -)$$
$$x_0 \to x_{11} \to x_{16} \to (P \bowtie I \bowtie E \bowtie C;\ -;\ -)$$
$$x_0 \to x_{10} \to (P;\ C \bowtie E \bowtie I;\ -) \to (P \bowtie I \bowtie E \bowtie C;\ -;\ -)$$
$$x_0 \to x_{11} \to (P;\ C \bowtie E \bowtie I;\ -) \to (P \bowtie I \bowtie E \bowtie C;\ -;\ -).$$

We now comment on the use of the "best-first" strategy of Section 5.3 for this example. Since there are only three steps in this problem, it is not necessary to

[6] $s(R)$ denotes the site where relation R is located.

x	$x(1)$	$x(2)$	$x(3)$	$l(x)$	$card(x)$	$C(x)$
0	P, C	I	E	0	1	0
1	$C, P \bowtie I$	-	E	1	1	100
2	C	$P \bowtie I$	E	1	1	200
3	C	-	$E, P \bowtie I$	1	1	200
4	$C, P \bowtie E$	I	-	1	1	500
5	C	$I, P \bowtie E$	-	1	1	1000
6	C	I	$P \bowtie E$	1	1	1000
7	$P, C, I \bowtie E$	-	-	1	1	130
8	P, C	$I \bowtie E$	-	1	2	100
9	$P, C \bowtie E$	I	-	1	1	100
10	P	$I, C \bowtie E$	-	1	1	100
11	P	I	$C \bowtie E$	1	1	50
12	C	$P \bowtie I \bowtie E$	-	2	2	160
13	$C, P \bowtie I \bowtie E$	-	-	2	1	130
14	$P \bowtie E \bowtie C$	I	-	2	2	100
15	-	$I, P \bowtie E \bowtie C$	-	2	1	150
16	$P, I \bowtie E \bowtie C$	-	-	2	1	110
17	P	$I \bowtie E \bowtie C$	-	2	2	100
18	$P \bowtie I, C \bowtie E$	-	-	2	3	200
19	$P \bowtie I$	$C \bowtie E$	-	2	6	150
f	$P \bowtie I \bowtie E \bowtie C$	-	-	3	3	110

Fig. 4. State space.

invoke this procedure recursively. A simple initial choice for step 1 would be: $Traj(1) = \{x_0, x_1, x_{13}, X_f\}$, whose cost is $R_f = 100 + 130 + 0 = 230$ (refer to Figure 5). This corresponds to having each new intermediate relation always located at site 1, the site where there is the largest number of tuples at the beginning. Then, x_4, x_5, and x_6 can be deleted from the state space immediately after their costs have been computed. This alone saves the calculation of 12 one-step optimal costs. Once $C(x_{13})$ has been computed, x_2 and x_3 can also be deleted because the updated R_f is now $130 + 0 = 130$. So if x_{12} is the choice for $x_{N-2}(2)$, then $T^-(x_{12})$ now contains only three states instead of the original seven. Further savings will occur at the subsequent steps.

8. INCLUDING SEMIJOIN STATE TRANSITIONS

8.1 Admissible Semijoin Transitions

We now remove the restriction imposed in Section 3.1 that only joins be admissible state transitions, and also include transitions consisting of semijoins. This time, since another join with B is required after $A \ltimes B$ has been done, B has to be kept in the new state. In other words, we now make the following deletions in the new state after a transition: (i) after $A \bowtie B$, remove both A and B from the new state, and (ii) after $A \ltimes B$, only remove A from the new state.

The q part in the complete state (x, q) must keep track of the new properties of the query to avoid idempotent transitions. Let $q_0 = A \bowtie B \bowtie C$, corresponding to the query graph: A —— B —— C, and let $x_0 = (A; B; C)$. Then, for $x_1 = (A'; B; C)$ with $A' = A \ltimes B$, the corresponding $q_1 = A' \bowtie B \bowtie C$ with the constraint that $A' \ltimes B = A'$. Observe also that the strategy sets Γ_S and Γ of

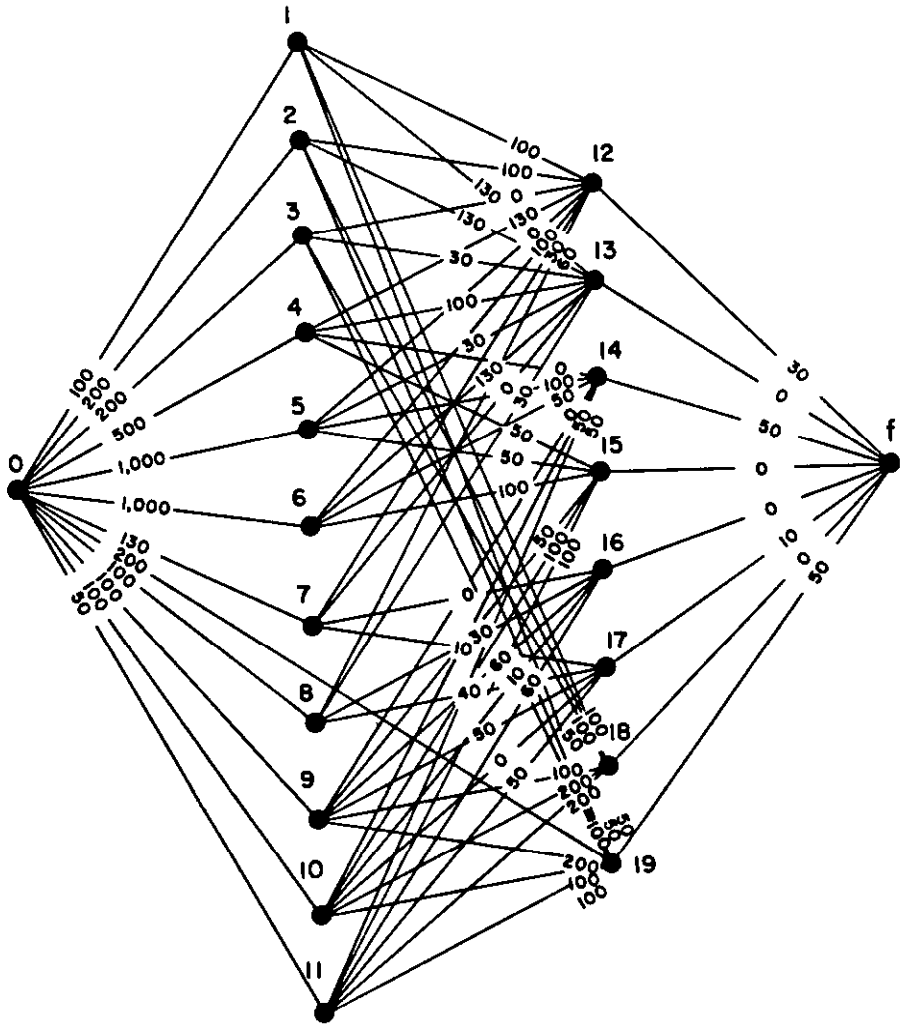

Fig. 5. State trajectories.

Section 4 (for step II of the algorithm) are not the same for semijoin and join state transitions. (They are clearly simpler for a semijoin.) In particular, join strategies based on elementary semijoin programs need not be included any longer because the intermediate semijoin states now appear explicitly in X.

We say that a relation has been fully *semijoin-reduced* when any further admissible semijoin on it is idempotent or brings no further deletions of tuples (cf. [14]). A state is fully semijoin-reduced when all the relations in it are fully semijoin-reduced.

Remark 8.1. In this paper we do not consider the reductive power of semijoins, as in [2–4, 14], but rather their efficiency, in terms of cost, as operations (state transitions) in distributed query processing. We need the concept of full

semijoin-reduction to determine a bound on the maximum number of state transitions before reaching X_f. For this purpose we invoke results from [2, 3].

In addition to the conditions in the definition of one-step transitions in Section 3.1, a one-step transition from state (x, q) involving $A \ltimes B$ with A and B relations in x is admissible iff:

(i) A is not fully semijoin-reduced;
(ii) A and B are linked in the query graph of q (observe that if there is more than one link between them, the semijoin is on all the attributes involved in these join clauses); and
(iii) A and B are located at different sites in x.

Furthermore, we require that (site of $(A \ltimes B)$ in the new state) = (site A in x). (These requirements are common in the literature; cf. [5, 8, 10, 11, 23, 25].)

Despite these restrictions, the number of semijoins that can be made in the process of solving q_0 is very high. The reason for this is that for general queries involving the three operations: projection, restriction, and (equality or inequality) join, the semijoin operation possesses no nice properties; in particular, it is not associative and rarely idempotent. Very long semijoin programs can be constructed to reduce a relation (see [8, pp. 141–5]).

Our interest is two-fold: (i) how to recognize fully semijoin-reduced states, and (ii) how to determine the maximum number of state transitions possible before such a state is reached, denoted NUM_{sj}. Once in such a state, a maximum of $N - 1$ additional transitions (corresponding to the $N - 1$ joins that must be done) necessarily yields a state in X_f. The answer to these questions depends on the form of q_0. For our purposes, we distinguish two categories of queries:

(1) those for which full semijoin-reduction can be identified syntactically, and for which NUM_{sj} is expressible in terms of N only;
(2) those for which full semijoin-reduction cannot be identified syntactically, and for which NUM_{sj} is of $O(m)$, where m is the number of *tuples* in some original relation referenced by the query.

The work in [2, 3] demonstrates that for equijoin queries,[7] the characterization between (1) and (2) is simple: (1) consists of the tree queries (roughly speaking, of the queries whose query graph has no cycles), whereas (2) regroups all other equijoin queries, denoted cyclic queries. (We refer the reader to [2, 3, 14] for the precise definition of tree query.) Queries with inequality joins have also been studied (see [4, 26]), but the results are quite different. In particular, full semijoin-reduction is possible for queries with "good cycles" [4]. For the sake of simplicity, we restrict our attention to equijoin queries and treat separately tree queries and cyclic queries in the next two sections.

Observe, however, that there is no other conceptual difference between the case we study in this section and the simpler case of the preceeding sections, as

[7] An equijoin query is a query that is a conjunction of join clauses, all the joins being on equality conditions (i.e., equijoins). They can be augmented by target lists and clauses involving constants, but these latter clauses should not be treated as links in the query graph, but instead separately by the restriction operation. They are the type of queries considered in [1, 10, 11, 15], for example.

far as the solution framework that we have proposed is concerned. Once the state space X has been constructed according to the above restrictions for admissible semijoins, the rest of the solution is exactly as described in Sections 3 to 6, and all the results there still hold. The only difference is that the maximum number of steps to reach X_f is no longer $N - 1$.

Now, the important fact is that the state-transition model is general enough to encompass and generalize many algorithms based on semijoin programs, in particular [5, 10, 11, 23]. All the strategies that the SDD-1 algorithm [5, 23] can reach and all the "correct nonredundant semijoin programs" for chain and tree queries of [10, 11] correspond to state trajectories in our framework, since all their intermediate steps are states in X. (The same cannot be said about the main algorithm of [1], because it can yield strategies containing more than one semijoin with the same relation, each one on a different attribute.)

8.2 Case of Tree Queries

Tree queries are simpler to analyze than cyclic ones due to the following lemma. (Without loss of generality, we assume a cycle-free query graph.)

LEMMA 8.1. *Let q_0 be a tree query referencing N original relations in a distributed database. Then the maximum number of state transitions to reach a state in X_f from x_0 is bounded above by $(N + 1)(N - 1)$.*

PROOF. The worst case occurs when each of the N original relations is located at a different site. Each such relation will be fully semijoin-reduced after $N - 1$ semijoins ([2], Theorem 1; [3], Theorem 1). Hence, with only one new semijoin at each transition (and this is always possible), after $N(N - 1)$ transitions, all the relations will have been fully semijoin-reduced, and any further admissible semijoin will be idempotent (cf. [11], Theorem 2). After that stage, $q_0(x_0)$ can be obtained after $N - 1$ joins. Clearly, interleaving join and semijoin transitions cannot result in more steps before $q_0(x_0)$ will be reached □

The crucial fact here is that the above upper bound only depends on the *number* of relations in the query, and not on the particular relations themselves. Hence, the results in Sections 3 to 5 can be directly applied to tree queries. Another interpretation to this fact is that in the case of tree queries, the semijoin operation can be viewed as possessing syntactic properties that make it simple to determine when a new semijoin is idempotent.[8] This is best illustrated by means of an example.

Example 8.1. Consider the tree query q_0: R —— S —— D, and let $x_0 = (R; S; D)$. First, observe that from the restrictions on admissible semijoins in Section 8.1, there is no ambiguity in writing $S \ltimes R \ltimes D$; it can only mean $(S \ltimes R) \ltimes D$. Then, it can be shown that

(i) $R \ltimes S \ltimes R = R \ltimes S$;
(ii) $(R \ltimes S \ltimes D) \ltimes (S \ltimes D \ltimes R) = R \ltimes S \ltimes D$;
(iii) $(R \ltimes S) \bowtie (S \ltimes D \ltimes R) = R \bowtie (S \ltimes D)$.

[8] These syntactic properties are used in [10, 11] to prune the set of semijoin reducer programs (which corresponds to eliminating some state trajectories in our model), but this pruning depends on the specific cost model considered in these references (affine in amount of data moved).

x	$x(1)$	$x(2)$	$x(3)$	$l(x)$	$card(x)$	$T^{+}(x)$
0	R	S	D	0	1	(1-2-3-4-17-18-19-20)
1	$R \ltimes S$	S	D	1	1	(5-6-7-17-18-26-27)
2	R	$S \ltimes R$	D	1	1	(6-8-11-17-18-35-37)
3	R	$S \ltimes D$	D	1	1	(8-9-13-19-20-34-36)
4	R	S	$D \ltimes S$	1	1	(5-9-10-19-20-28-29)
5	$R \ltimes S$	S	$D \ltimes S$	2	1	(14-15-26-27-28-29)
6	$R \ltimes S$	$S \ltimes R$	D	2	1	(12-21-17-18-41-42)
7	$R \ltimes S$	$S \ltimes D$	D	2	1	(12-13-14-26-27-34-35)
8	R	$S \ltimes R \ltimes D$	D	2	1	(22-23-34-35-36-37)
9	R	$S \ltimes D$	$D \ltimes S$	2	1	(16-24-19-20-43-44)
10	R	$S \ltimes R$	$D \ltimes S$	2	1	(11-15-16-28-29-36-37)
11	R	$S \ltimes R$	$D \ltimes S \ltimes R$	3	1	(21-23-32-33-36-37)
12	$R \ltimes S$	$S \ltimes R \ltimes D$	D	3	1	(22-30-34-35-41-42)
13	$R \ltimes S \ltimes D$	$S \ltimes D$	D	3	1	(22-24-34-35-38-39)
14	$R \ltimes S$	$S \ltimes D$	$D \ltimes S$	3	1	(24-25-26-27-43-44)
15	$R \ltimes S$	$S \ltimes R$	$D \ltimes S$	3	1	(21-25-28-29-41-42)
16	R	$S \ltimes R \ltimes D$	$D \ltimes S$	3	1	(23-31-36-37-43-44)
17	$R \bowtie S$	-	D	3	2	f
18	-	-	$D, R \bowtie S$	3	1	f
19	$R, D \bowtie S$	-	-	3	1	f
20	R	$D \bowtie S$	-	3	2	f
21	$R \ltimes S$	$S \ltimes R$	$D \ltimes S \ltimes R$	4	1	(30-32-33-41-42)
22	$R \ltimes S \ltimes D$	$S \ltimes R \ltimes D$	D	4	1	(40-34-35-47-48)
23	R	$S \ltimes R \ltimes D$	$D \ltimes S \ltimes R$	4	1	(40-36-37-45-46)
24	$R \ltimes S \ltimes D$	$S \ltimes D$	$D \ltimes S$	4	1	(31-38-39-43-44)
25	$R \ltimes S$	$S \ltimes D \ltimes R$	$D \ltimes S$	4	1	(30-31-41-42-43-44)
26	$R \ltimes S, S \bowtie D$	-	-	4	1	f
27	$R \ltimes S$	$S \bowtie D$	-	4	2	f
28	$R \bowtie S$	-	$D \ltimes S$	4	2	f
29	-	-	$D \ltimes S, R \bowtie S$	4	1	f
30	$R \ltimes S$	$S \ltimes R \ltimes D$	$D \ltimes S \ltimes R$	5	1	(40-41-42-45-46)
31	$R \ltimes S \ltimes D$	$S \ltimes D \ltimes R$	$D \ltimes S$	5	1	(40-43-44-47-48)
32	$R \bowtie S$	-	$D \ltimes S \ltimes R$	5	2	f
33	-	-	$D \ltimes S \ltimes R, R \bowtie S$	5	1	f
34	$R \bowtie (S \ltimes D)$	-	D	5	2	f
35	-	-	$D, R \bowtie (S \ltimes D)$	5	1	f
36	R	$D \bowtie (S \ltimes R)$	-	5	2	f
37	$R, D \bowtie (S \ltimes R)$	-	-	5	1	f
38	$R \ltimes S \ltimes D, D \bowtie S$	-	-	5	1	f
39	$R \ltimes S \ltimes D$	$D \bowtie S$	-	5	2	f
40	$R \ltimes S \ltimes D$	$S \ltimes R \ltimes D$	$D \ltimes S \ltimes R$	6	1	(45-46-47-48)
41	$R \ltimes S$	$D \bowtie (S \ltimes R)$	-	6	2	f
42	$R \ltimes S, D \bowtie (S \ltimes R)$	-	-	6	1	f
43	$R \bowtie (S \ltimes D)$	-	$D \ltimes S$	6	2	f
44	-	-	$D \ltimes S, R \bowtie (S \ltimes D)$	6	1	f
45	$R \bowtie (S \ltimes D)$	-	$D \ltimes S \ltimes R$	7	2	f
46	-	-	$D \ltimes S \ltimes R, R \bowtie (S \ltimes D)$	7	1	f
47	$R \ltimes S \ltimes D, D \bowtie (S \ltimes R)$	-	-	7	1	f
48	$R \ltimes S \ltimes D$	$D \bowtie (S \ltimes R)$	-	7	2	f
f	$R \bowtie S \bowtie D$	-	-	8	3	-

Fig. 6. Example 8.1.

Using these and similar results, the state space for this example can be constructed in a straightforward manner. It is given in Figure 6 (we have used equivalence classes for simplicity). In that figure we also list the elements of each $T^+(\mathbf{x})$ set. In order to keep this example simple, we have not performed any semijoin transition when the answer could be reached in one more join. For the last join, one can always include elementary semijoin programs in Γ_S instead of explicitly allowing a semijoin transition. Observe the advantage of dynamic programming in this example, where $card(X) = 50$, while Figure 6 indicates that there are more than 740 trajectories between x_0 and X_f.

The same example is treated in [8, p. 146–7], using the SDD-1 algorithm. The solution given by that algorithm corresponds to the following state trajectory in our model: x_0, x_3, x_8, $\mathbf{x}_{34}$, X_f.

8.3 Case of Cyclic Queries

There is an extra conceptual difficulty in handling cyclic queries. The maximum number of semijoins that can be done before full semijoin-reduction is attained is of the order of the number of tuples in some relation in the query. Thus, in our framework, there is no upper bound on the maximum number of steps that can be expressed as a function of N.

Essentially the reason for so many steps is that, in contrast to the case of tree queries, semijoin programs for cyclic queries cannot be syntactically examined for idempotence, as was possible in Example 8.1. In general, examining the tuples in two relations is necessary to determine if semijoins between them will reduce one of the two.

Since it is necessary to recognize when each relation cannot be further semijoin-reduced to determine which states X can be limited, the state space for such queries depends on the particular tuples in the relations referenced by q_0, and not only on q_0 and x_0. This suggests that it may be impractical to solve the problem in such generality.

For this reason, we suggest below a list of heuristics that can be used to reduce $card(X)$, resulting in the determination of a possibly suboptimal solution. (In any case, long semijoin reductions are unlikely to yield optimal trajectories.)

(i) Impose a bound on the maximum number of semijoins, based on the size of an original relation after each reduction (in practice, on the estimate of this size). If there are n links in the query graph and this bound is $2n$, then the state space will be large enough to contain any strategy obtainable by the SDD-1 algorithm. Since a maximum of $2n$ semijoins are considered in the first iteration of that algorithm, it has a maximum of $2n$ iterations.

(ii) Allow only some semijoin programs for each original relation, based for example on the size of these relations and on the query graph.

(iii) Transform the cyclic query into a tree query (see [25] for a list of some methods that have been proposed). For example, break each cycle in the query graph of q_0 by imposing a specific join for the first state transition. After that step, the query becomes a tree query.[9] (Choose the cheapest join in each cycle, or exhaustively solve the problem for each possible combination of choices.)

[9] This corresponds to the "relation-merging algorithm" mentioned in [25].

(iv) Solve each cycle in the query graph by considering only joins as state transitions, and then solve the resulting tree query where each cycle is now considered an original relation.

9. REDUNDANT INITIAL MATERIALIZATIONS

We now discuss the effect of beginning with an initial materialization that may contain more than one copy of each original relation. Since a join or semijoin of a relation with itself is not an admissible state transition, the fact that x_0 is redundant brings no complication, provided,

(1) for a state transition to be admissible, it must be admissible for each possible selection of copies of the original relations involved in it;
(2) in the computation of the function c in part II of the algorithm, an extra minimization is carried over all possible selections of copies for the original relations involved in the state transition; and
(3) when the rules for state transitions require the deletion of a relation from a state, all copies of that relation are deleted.

Proceeding in this manner is roughly equivalent to solving the same problem for each possible irredundant x_0, and then taking the minimum among these optimal solutions. This is exact if only join transitions are allowed, but when semijoin transitions are also included, the same copy need not be used each time an original relation is part of a transition, and thus (1)–(3) permit more generality.[10] (1)–(3) are advantageous because the extra minimization is brought at the individual one-step state transition level, thus significantly simplifying the task.

In conclusion, under conditions (1)–(3), all the results in this paper remain valid when x_0 is redundant.

10. CONCLUSION

We have presented a state transition model for the complete solution to the problem of optimizing the processing of a query in a distributed database system, when the join and semijoin operations are taken as the unit step in the sequence of operations. The cost model can be as general as desired. By defining a state space to parametrize the evolution of the processing, the problem can be separated into two stages.

In the first stage all one-step state transitions must be optimized. This problem has been addressed in the literature, and many different strategies can be used for the optimization. The possibility of choosing among various copies of the relations can also be included at this stage. We believe that the problem of the estimation of the costs that is central to this stage is no more difficult than in the other works on distributed query processing in the literature.

Then, in the second stage, dynamic programming is applied over the state space to determine the minimum-cost sequence of operations (state trajectory) yielding the answer to the query. This separation is an important feature, because

[10] This distinction is irrelevant in the site-uniformity case.

by properly defining the concept of state transitions, we are able to incorporate the possibility of parallel processing without any further modifications.

We do not believe that the size of the state space hinders the practicality of our algorithm. Experiences from query optimization for centralized databases indicate that the cost of computing an optimal solution is often overestimated, while the benefit is underestimated. Our premise is that this may also be true for distributed databases. Moreover, the savings from dynamic programming over an exhaustive search well compensates for the extra work in constructing the state space. Concerning the case of join state transitions only, we believe that our algorithm, by explicitly defining a state has computational advantages over the algorithm in [18], which also uses a form of dynamic programming.

In the case of both join and semijoin state transitions, we allow any sequence of these two operations, which is considerably more general than the popular reduction-phase/assembly-phase strategy in the literature. In fact, there is no guarantee that semijoin-reducer programs will be optimal. Moreover, not executing all the joins at the same site, but rather in a distributed fashion, may render additional semijoins profitable. Of course, the optimization requires more work in this case, although, apart from the problem of the construction of the state space for cyclic queries, it is only computationally more difficult, not conceptually so. We stress that the use of dynamic programming is significant here, especially in the case of tree queries, due to the large "fan-out" of the state trajectories in the state space (see Example 8.1).

Another important benefit of the state parameterization is that it provides a precise framework into which many additional refinements can be incorporated. For example, clever strategies concerning the computation of the function C over the state space can improve on the systematic recursive approach of part III of the algorithm. The "best-first" strategy of Section 5.3 is an example of such a modification. These strategies require a minimal amount of additional work (for example, the computation of an initial upper bound for the optimal cost), but the rewards can be significant (cf. Section 7).

Our state transition model can indeed be applied to query optimization for a centralized database. The development is the same as in this paper, with the restriction that $M = 1$. We believe that in that case it is worth refining the state space by considering as state information the ordering of the tuples in the intermediate results (see Remark 4.1). (Such orderings are explicitly considered by the optimizers of systems R [21] and R^* [18].) Clearly, only the existence of an order on the attributes which are part of a subsequent join are of interest. The case of centralized databases is discussed in more detail in [17].

Among the areas of interest for future work, we mention:

(i) the appropriate selection of the distributed join strategies to include in Γ_S of (4.1);
(ii) the determination of efficient semijoin strategies for cyclic queries, alleviating the inconvenience of long semijoin sequences;
(iii) the study of other ways of improving on the basic dynamic programming algorithm; and

(iv) the use of a similar state-transition approach in the case of recursive queries (see [16] for recent work on conditions for the existence of bounds on the number of steps in this case).

REFERENCES

1. APERS, P. G. M., HEVNER, A. R., AND YAO, S. B. Optimization algorithms for distributed queries. *IEEE Trans. Softw. Eng. SE-9*, 1 (Jan. 1983), 57–68.
2. BERNSTEIN, P. A., AND CHIU, D.-M. Using semijoins to solve relational queries. *J. ACM 28*, 1 (Jan. 1981), 25–40.
3. BERNSTEIN, P. A., AND GOODMAN, N. Power of natural semijoin. *SIAM. J. Comput. 10*, 4 (Nov. 1981), 751–771.
4. BERNSTEIN, P. A., AND GOODMAN, N. The power of inequality semijoins. *Inf. Syst. 6*, 4 (1981), 255–265.
5. BERNSTEIN, P. A., GOODMAN, N., WONG, E., REEVE, C., AND ROTHNIE, J. Query processing in a system for distributed databases (SDD-1). *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 602–625.
6. BLASGEN, M. W., AND ESWARAN, K. P. On the evaluation of queries in a relational database system. Res. Rep. RJ 1745, IBM Research Laboratory, San Jose, Calif., Apr. 1976.
7. CAREY, M. J., AND LU, H. Some experimental results on distributed join algorithms in a local network. Computer Science Rep. 587, Univ. of Wisconsin. Also in the *Proceedings of the 11th Conference on Very Large Data Bases* (Stockholm, 1985).
8. CERI, S., AND PELAGATTI, G. *Distributed Databases—Principles and Systems*. McGraw-Hill Computer Science Series, New York, 1984.
9. CHEN, A., AND LI, V. Optimizing star queries in distributed database systems. In *Proceedings of the 10th Conference on Very Large Data Bases* (1984), 429–438.
10. CHIU, D.-M., BERNSTEIN, P. A., AND HO, Y.-C. Optimizing chain queries in a distributed database system. *SIAM J. Comput. 13*, 1 (Feb. 1984), 116–134.
11. CHIU, D.-M., AND HO, Y.-C. A methodology for interpreting tree queries into optimal semijoin expressions. In *Proceedings of the 1980 ACM-SIGMOD Conference* (Santa Monica, Calif., May 1980), ACM, New York, 169–178.
12. CHU, W. W., AND HURLEY, P. Optimal query processing for distributed database systems. *IEEE Trans. Comput. C-31*, 9 (Sept. 1982), 835–850.
13. EPSTEIN, R., STONEBRAKER, M., AND WONG, E. Distributed query processing in a relational database system. In *Proceedings of the 1978 ACM-SIGMOD Conferece* (Austin, Tex., May 1978), ACM, New York, 169–180.
14. GOODMAN, N., AND SHMUELI, O. Tree queries: A simple class of relational queries. *ACM Trans. Database Syst. 7*, 4 (Dec. 1982), 653–677.
15. HEVNER, A. R., AND YAO, S. B. Query processing in distributed database systems. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 177–187.
16. IOANNIDIS, Y. A time bound on the materialization of some recursively defined views. In *Proceedings of the 11th Conference on Very Large Data Bases* (Stockholm, 1985).
17. LAFORTUNE, S. Distributed information and distributed control: Cases from stochastic systems and database mangement. Ph.D. dissertation, Univ. of California, Berkeley, May 1986.
18. LOHMAN, G. M., MOHAN, C., HAAS, L. M., LINDSAY, B. G., SELINGER, P. G., WILMS, P. F., AND DANIELS, D. Query processing in *R**. Res. Rep. RJ 4272, IBM Research Laboratory, San Jose, Calif., Apr. 1984.
19. RICH, E. *Artificial Intelligence*. McGraw-Hill, New York, 1983.
20. SELINGER, P. G., AND ADIBA, M. Access path selection in distributed database management systems. Res. Rep. RJ 2883, IBM Research Laboratory, San Jose, Calif., Aug. 1980.
21. SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. Access path selection in a relational database management system. Res. Rep. RJ 2429, IBM Research Laboratory, San Jose, Calif., Jan. 1979.
22. VALDURIEZ, P., AND GARDARIN, G. Join and semijoin algorithms for a multiprocessor database machine. *ACM Trans. Database Syst. 9*, 1 (Mar. 1984), 133–161.

23. WONG, E. Retrieving dispersed data from SDD-1: A system for distributed databases. In *Proceedings of the 2nd Berkeley Workshop on Distributed Data Management and Computer Networks* (Lawrence Berkeley Laboratory, May 25-27, 1977), 217-235.
24. WONG, E. Dynamic rematerialization: Processing distributed queries using redundant data. *IEEE Trans. Softw. Eng. SE-9*, 3 (May 1983), 228-232.
25. YU, C. T., AND CHANG, C. C. Distributed query processing. *ACM Comput. Surv. 16*, 4 (Dec. 1984), 399-433.
26. YU, C. T., AND OZSOYOGLU, M. Z. On determining tree query membership of a distributed query. *Can. J. Oper. Res. Inf. Process. 22*, 3 (Aug. 1984), 261-268.
27. YU, C. T., OZSOYOGLU, M. Z., AND LAM, K. Distributed query optimization for tree queries. *J. Comput. Syst. Sci. 29* (1984), 409-445.

Received September 1985; revised January 1986; accepted February 1986