INTERLEAVING A JOIN SEQUENCE WITH SEMIJOINS IN DISTRIBUTED QUERY PROCESSING

Ming-Syan Chen and Philip S. Yu
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract

In distributed query processing, the conventional approach to reduce the amount of data transmission is to first apply a sequence of semijoins as "reducers" and then ship the resultant relations to the final site to carry out the join operations. Recently, it has been shown that the approach of applying a combination of joins and semijoins as reducers can lead to substantially larger reduction on data transmission required. In this paper, we develop an efficient heuristic approach to determine an effective sequence of semijoin and join reducers. Semijoins whose execution will reduce the amount of data transmission required to perform a join sequence are termed *beneficial semijoins* for that join sequence. Note that beneficial semijoins include the conventional profitable semijoins and the *gainful semijoins* that are not profitable themselves but become beneficial due to the inclusion of join reducers. This type of dependency between semijoin and join reducers complicates the identification of beneficial semijoins and the ordering in the reducer sequence. In this paper, we first obtain a sequence of join reducers and map it into a join sequence tree. In light of the join sequence tree, we derive important properties of beneficial semijoins. These properties are then applied to develop an efficient algorithm to determine the beneficial semijoins which can be inserted into the join sequence. Examples are also given to illustrate this approach. Our results show that the approach of interleaving a join sequence with beneficial semijoins are not only efficient but also effective in reducing the total amount of data transmission required to process distributed queries.

Index Terms: Distributed query processing, gainful semijoins, beneficial semijoins, join sequence tree, reducible set.

1 Introduction

In a distributed relational database system, the processing of a query involves data transmission among different sites via a computer network. As pointed out in [27], the processing of a distributed query in such a system is composed of the following three phases: (1) *local processing phase* which involves all local processing such as selections and projections, (2) *reduction phase* where a sequence of semijoins is used to reduce the size of relations, and (3) *final processing phase* in which all resulting relations are sent to the site where the final query processing is performed. The objective taken in this context is mainly to reduce the communication cost required for data transmission [5]. Significant research efforts have been focused on the problem of reducing the amount of data transmission required for phases (2) and (3) of distributed query processing [1]-[12] [14]-[18] [21] [22] [25] [26]. The semijoin operation especially has received considerable attention and been extensively

studied in the literature. It has been proved that a tree query can be fully reduced by using semijoin [2], and there has been much research reported in optimizing semijoin sequences to process certain tree queries, such as star and chain queries [7] [10]. However, the determination of an optimal semijoin sequence for general tree queries has been proved to be NP-hard [23]. For general query graphs with cycles, even with one join attribute, the problem of finding an optimal strategy to minimize the data transmission cost has also been proved to be NP-hard [14].

In addition to semijoins, join operations can also be used as reducers in processing distributed queries [8] [9] [17] [19]. As shown in [8] [9] and to be illustrated later, judiciously applying join operations as reducers can further reduce the amount of data transmission required. Moreover, as pointed out in [8], the approach of combining join and semijoin operations as reducers can result in more beneficial semijoins due to the inclusion of joins as reducers. (Such semijoins are referred to as *gainful semijoins* in [8].) In addition, this approach can reduce the communication cost further by taking advantage of the removability of pure join attributes[1]. For simplicity, both the profitable semijoins and the gainful semijoins in [8] are called *beneficial semijoins* in this paper. In [9], it is proved that the problem of determining the optimal sequence of join operations for a given query graph is of exponential complexity, thus justifying the need to apply heuristic approaches to deal with this problem. Also, it is shown in [9] that by mapping the problem of determining a sequence of join reducers for a query into that of finding a specific type of cut set for the query graph[2], one can develop efficient heuristic algorithms of polynomial time complexity for tree and general query graphs respectively. However, the issue of identification of gainful semijoins was not addressed in [9] where the semijoin sequences are assumed to be given and applied prior to the join reducers. Note that as gainful semijoins depend upon subsequent join and semijoin operations, they can not be determined in isolation as profitable semijoins. Consequently, despite its importance, the problem of finding an ordered sequence of join and semijoin reducers for distributed query processing was not fully explored, and in fact, there is no efficient algorithm proposed thus far for such a problem. This is mainly due to the inherent difficulty of this problem, since the dependency between semijoin and join reducers significantly complicates the identification of beneficial semijoins as well as the ordering in the reducer sequence. In view of this fact, we in this paper focus on the issue of determining the beneficial semijoins (including profitable and gainful semijoins) and the proper ordering to insert the semijoins determined into the join sequence to form a sequence of join and semijoin reducers for distributed query processing. To the best of our knowledge, no prior work has either explored the theoretical aspects of, or developed algorithms for such an approach. This fact distinguishes our work from others.

We shall first obtain a join sequence and then map the sequence of joins into a join sequence tree. For example, consider the query graph [5] in Fig. 1. Assume R_3 is in the site where the final results are needed. The join sequence tree[3] for a join sequence $R_1 \Rightarrow R_5$, $R_5 \Rightarrow R_2$, $R_4 \Rightarrow R_2$ and $R_2 \Rightarrow R_3$ can be found in Fig. 2. In light of the structure of a join sequence tree, we can derive important properties of beneficial semijoins for the join sequence tree. These properties will then be applied to develop an efficient algorithm to determine beneficial semijoins for the join sequence. It is worth mentioning that the conventional approach of sending all the relations to the final site in phase (3) of the query processing is corresponding to the join sequence tree in Fig. 3, and thus a special case of our study. Examples will be given to illustrate our results. It can be seen that the approach to determine beneficial semijoins and interleave a join sequence with beneficial semijoins is not only efficient but also effective in reducing the total amount of data transmission required to

[1] Pure join attributes are those which are used in join predicates but not part of the output attributes.

[2] This type of cut set is termed complete and feasible (CF) set of cuts in [9].

[3] The formal definition of a join sequence tree is given in Section 3.

process a distributed query, thus making the approach of using a combination of joins and semijoins as reducers more attractive.

This paper is organized as follows. The notation and definitions required are given in Section 2.1 and some facts of using a combination of join and semijoin reducers are given in Section 2.2. In Section 3, we first introduce the mapping to obtain a join sequence tree, and then derive important properties for beneficial semijoins which are applied later to develop an algorithm for determining beneficial semijoins for a join sequence. Illustrative examples are presented in Section 4. This paper concludes with Section 5.

2 Preliminaries

The notation, definitions and assumptions required are stated in Section 2.1, and some properties and an example for the approach of combining joins and semijoins as reducers in query processing are presented in Section 2.2.

2.1 Notation, definitions and assumptions

As in most previous work in distributed databases [5] [27], we assume that the cost for executing a query can mainly be expressed in terms of the total amount of inter-site data transmission required. Also, it is assumed that a query is of the form of conjunctions of equi-join predicates and all attributes are renamed in such a way that two join attributes have the same attribute name if and only if they have a join predicate between them. To facilitate our presentation, we assume that relations referenced in the query are located in different sites[4]. When multiple copies of a relation exist, we assume that one copy has been preselected.

A join query graph can be denoted by a graph G = (V, E), where V is the set of nodes and E is the set of edges. An edge connecting two nodes n_i and n_j is denoted by (n_i,n_j), and said to be *incident* to n_i and n_j. Each node in a join query graph represents a relation. Two nodes are connected by an edge if there exists a join predicate on some attribute of the two corresponding relations. An edge (n_i,n_j) in a graph G is being *shrunken* if (n_i,n_j) is removed from the graph and n_i and n_j are merged together. Notice that when a join operation between the two relations corresponding to nodes n_i and n_j in a given query graph G is carried out, we can obtain the resulting query graph by shrinking the edges between n_i and n_j to represent the resulting relation from the join operation. Also we use $N_T(\mathrm{G})$ to denote the number of tuples in the relation resulting from the query graph G.

We use $|K|$ to denote the cardinality of a set K. Let w_A be the width of an attribute A and w_{R_i} be the width of a tuple in R_i. The size of the total amount of data in R_i can then be denoted by $w_{R_i}|R_i|$. For notational simplicity, we use $|A|$ to denote the cardinality of the domain of an attribute A. Define the selectivity $\rho_{i,a}$ of attribute A in R_i as $\frac{|R_i(A)|}{|A|}$, where $R_i(\mathrm{A})$ is the set of distinct values for the attribute A in R_i. We use $R_i - A \rightarrow R_j$ to mean a semijoin from R_i to R_j on attribute A, in which R_i is called the *reducer* and R_j is called the *reducee* of the semijoin. Note that the reduction of the relation R_j by the semijoin $R_i - A \rightarrow R_j$ is proportional to the reduction of $R_j(\mathrm{A})$. The estimation of the size of the relation reduced by a semijoin is thus similar to estimating the reduction of projection on the semijoin attributes. After the semijoin $R_i - \mathrm{A} \rightarrow R_j$, the cardinality of R_j can be estimated as $|R_j|\rho_{i,a}$. A semijoin $R_i - A \rightarrow R_j$, is called *profitable*

[4]Note that if some relations referenced by a query are located in the same site, some local processing can be performed first and the query is modified accordingly before the proposed scheme is applied.

if its cost of sending R_i(A), $w_A|R_i(A)| = w_A|A|\rho_{i,a}$, is less than its benefit, $w_{R_j}|R_j| - w_{R_j}|R_j|\ \rho_{i,a} = w_{R_j}|R_j|(1-\rho_{i,a})$, where $w_{R_j}\ |R_j|$ and $w_{R_j}|R_j|\rho_{i,a}$ are respectively the sizes of R_j before and after the semijoin. Note that, instead of directly sending R_i(A) to R_j, there can be different methods to carry out the semijoin operation $R_i -$ A $\rightarrow R_j$. For example, in the case that $|R_j(A)|$ is much less than $|R_i(A)|$, R_j can send R_j(A) to R_i first, and then R_i, instead of sending R_i(A) to R_j, sends to R_j a bit vector to indicate the matching attributes so as to reduce the cost of the semijoin[5]. In view of this, we can use the minimal cost of the applicable methods as the cost of the semijoin and determine if a semijoin is profitable[6]. Nevertheless, as can be seen later, the results and the algorithm developed in this paper do not rely on any particular semijoin method. Without loss of generality, as in most prior work [27], we use $w_A|R_i(A)|$ to denote the cost of a semijoin $R_i - A \rightarrow R_j$. To simplify the notation, we use $R_i \rightarrow R_j$ to mean a semijoin from R_i to R_j in the case that the semijoin attribute does not have to be specified. Also, the notation $R_i \Rightarrow R_j$ is used to mean that R_i is sent to the site of R_j and a join operation is performed with R_j there. We use R_i' to denote the resulting relation after some reducers (joins or semijoins) are applied to an original relation R_i. In addition, we assume that the values of attributes are uniformly distributed over all tuples in a relation and that the values of one attribute are independent of those in another attribute.

Suppose R_1 has two attributes A and B. The problem of estimating the cardinality of R_1 projected on the non-semijoin attribute B after the semijoin operation R_2 −A $\rightarrow R_1$, where $|R_1|$=n, $|R_1(B)|$=m and $|R_1|\rho_{2,a}$=k, has been studied and can be described by the following combinatorial problem: "There are n balls with m different colors. Each ball has one color and the m colors are uniformly distributed over the n balls. Find the expected number of colors if k balls are randomly selected from the n balls." Denote the expected number of colors of the k selected balls as g(m,n,k). Then, as pointed out in [24], g(m,n,k) can be formulated as follows.

$$g(m,n,k) \ = \ m[1 - \prod_{i=1}^{k}(\frac{\frac{n(m-1)}{m} - i + 1}{n - i + 1})] \tag{1}$$

As shown in [2], Eq.(1) can be approximated as below.

$$g(m,n,k) \ \simeq \ \begin{cases} m, & \text{for } m < \frac{k}{2}, \\ k, & \text{for } k < \frac{m}{2}, \\ \frac{(m+k)}{3}, & \text{otherwise.} \end{cases} \tag{2}$$

It can be seen that when $|R_1(B)|$=m is much less than $|R_1|\rho_{2,a}$=k, the cardinality of R_1(B) remains approximately the same after the semijoin R_2 −A $\rightarrow R_1$. Thus, as in most prior work [1] [3], we assume in this paper the cardinality of a non-semijoin attribute remains the same after a semijoin operation to simplify our discussion. In addition, to facilitate our presentation for the join sequence tree later, it is necessary to introduce the structure of a tree. A tree is a connected acyclic graph [13]. If every edge in a tree is directed and all the arrows in edges are away from a single node, the directed tree is called a rooted tree and that single node is called the root of the tree. Note that a rooted tree can be viewed as a partial order set. We denote $n_i \geq n_j$ if there is a path along the arrows in the tree from n_i to n_j. In such a case, node n_j (n_i) is called an offspring (ancestor) of n_i (n_j). We use $n_i > n_j$ to mean $n_i \geq n_j$ and $n_i \neq n_j$. Use T_{n_i} to denote the subtree formed by n_i and its offspring in a rooted tree T, and let S(T_{n_i}) be the set of nodes in T_{n_i}, i.e., S(T_{n_i})=$\{n_j | n_i \geq n_j, n_j \in S(T)\}$. Define the *lowest common ancestor* of two nodes n_i and n_j in a rooted tree, denoted by $n_i \vee n_j$, to be the node that is an ancestor of n_i and n_j and none of its

[5]Such a method was pointed out by an anonymous referee.

[6]This is also true in determining if a semijoin is beneficial later.

offspring is an ancestor of n_i and n_j. For example, for a rooted tree in Fig. 4a, T_{n_2} is given in Fig. 4b, and S(T_{n_2})= $\{n_2, n_3, n_4, n_5\}$. Also, $n_4 \vee n_5 = n_2$ and $n_5 \vee n_7 = n_1$ in Fig. 4a. In addition, when $n_i \geq n_j$ in a rooted tree T, we use P(n_i,n_j) to denote the set of nodes that are on the path from n_i to n_j excluding n_i, i.e., P(n_i,n_j)= $\{n_k | n_i > n_k \geq n_j \text{and} i \neq k, \forall n_k \in S(T)\}$. In the rooted tree in Fig. 4a, P(n_2,n_4)=$\{n_3, n_4\}$ and P(n_1,n_5)=$\{n_2, n_5\}$.

2.2 Inclusion of join operations as reducers in query processing

In this section, we shall first describe in Section 2.2.1 the method to estimate the effect of a set of join operations on a query graph, which has been formulated in [8]. In Section 2.2.2 we describe the concepts of gainful semijoins and pure join attributes which occur with the use of join operations as reducers in query processing. As pointed out in [8], the two concepts are very useful in further reducing the amount of data transmission required for query processing, thus increasing the applicability of join reducers. An illustrative example for the inclusion of joins as reducers is given in Section 2.2.3.

2.2.1 determination of the effect of join operations

To determine the effect of a join operation specified by a query graph, the following theorem was developed in [8].

Theorem 1 [8]: Let G=(V,E) be a join query graph. G_B=(V_B, E_B) is a connected subgraph of G. Let R_1, R_2,..., R_p be the relations corresponding to nodes in V_B, A_1, A_2,..., A_q be the distinct attributes associated with edges in E_B and m_i be the number of different nodes (relations) that edges with attribute A_i are incident to. Suppose R^* is the relation resulting from the join operations between relations in G_B and $N_T(G_B)$ is the expected number of tuples in R^*. Then

$$N_T(G_B) = \frac{\prod_{i=1}^{p} |R_i|}{\prod_{i=1}^{q} |A_i|^{m_i - 1}}. \quad (3)$$

For the example query in Fig. 5a, the expected number of tuples in the resulting relation is $\frac{|R_1||R_2||R_3||R_4|}{|A|^2|B||C||D|}$. It can be verified that in the case that R_i, 1≤i≤ $|V_B|$, are those resulting from some semijoins, a similar result still holds, but the cardinalities of the corresponding domains have to be modified accordingly. For example, suppose that R_i, 1≤i≤4, in Fig. 5a are those after semijoins R_2 $-\text{A} \rightarrow R_1$ and R_3 $-\text{B} \rightarrow R_1$ have been performed. The expected number of tuples in the resulting relation is thus estimated as $\frac{|R_1\prime||R_2||R_3||R_4|}{\rho_{2,a}|A|^2\rho_{3,b}|B||C||D|}$. Moreover, the estimated cardinality of the resulting relation is independent of the sequence in which those join operations are performed. For example, consider the query in Fig. 5a. Suppose V_{R_1}=$\{R_1, R_4\}$ and V_{R_2}=$\{R_2, R_3\}$. The corresponding G^* is given in Fig. 5b. Then we have $|R_1^*| = \frac{|R_1||R_4|}{|D|}$ and $|R_2^*| = \frac{|R_2||R_3|}{|A||C|}$. It can be verified that N_T(G) $= N_T(G^*) = \frac{|R_1^*||R_2^*|}{|A||B|}$.

Notice that for a given query there are usually many potential sequences of reducers to perform the query. However, different sequences, though resulting in the same final relation, may involve different transmission costs since the intermediate relations in different sequences may have different sizes. In light of the results in Theorem 1, we can estimate the sizes of intermediate relations for each sequence and determine the cost of a sequence of join operations.

2.2.2 gainful semijoins and pure join attributes

The concepts of gainful semijoins and pure join attributes occur with the use of join operations. As pointed out earlier, the application of join operations as reducers may result in more beneficial

semijoins available. Those semijoins which become beneficial due to the use of join and semijoin reducers are termed gainful semijoins. An example for the gainful semijoin can be found in the following subsection. For convenience, both profitable semijoins and gainful semijoins are called beneficial semijoins in this paper[7]. Note that whether a semijoin is gainful or not depends on the subsequent reducer operations. Also, it can be verified that gainful semijoins are not only those that become profitable after the corresponding join operations are performed.

In addition, some join attributes in a query may not be part of the final answer. Therefore, after the corresponding join operations are performed we can remove some attributes which are not needed further to reduce the amount of data transmission required in the subsequent operations. Consider the following query as an example.

select A,C **from** R_1, R_2, R_3 **where** R_1.A = R_3.A and R_1.B = R_2.B and R_2.C = R_3.C.

In the above query, A and C are output attributes and B is a pure join attribute. Since B is not needed after the join between R_1(A,B) and R_2(B,C), we can remove attribute B by performing a projection of $R_2\prime$(A,B,C) on (A,C), where $R_2\prime$(A,B,C) is the relation resulting from joining R_1 and R_2. Note that the removal of pure join attributes is only available when joins are included as reducers since a pure join attribute can be removed only after the corresponding join operation has been performed.

2.2.3 example for the inclusion of joins as reducers

Consider the following query:

select B,D,F **from** R_1, R_2, R_3, R_4, R_5 **where** R_1.A = R_5.A and R_1.B = R_2.B and R_2.C = R_5.C, R_2.F = R_3.F and R_3.E = R_4.E and R_4.D = R_5.D.

The corresponding query graph and profile can be found in Fig. 1 and Table 1, respectively. Suppose R_3 is the site where the final results are needed.

Case1 : Using only semijoins as reducers for query processing.

From the data shown in Table 1, it can be seen that R_3 $-$F $\rightarrow$ R_2 and R_4 $-$D $\rightarrow$ R_5 are profitable semijoins and should be executed in phase (2). The costs required for the two semijoins are 450 and 700 respectively. After the execution of the two semijoins, the corresponding data in the profile of the query are changed. It can be verified that there is no profitable semijoin available thereafter. Then in phase (3), one has 3570+ 12900 +6200 +11298 =33968 units of transmission cost to send the remaining data in R_1, R_2, R_4 and R_5 to the final site where R_3 is located. The total transmission cost for phases (2) and (3) is 1150+33968 =35118.

Case2 : Using joins and semijoins as reducers for query processing.

When join operations are also used as reducers in distributed query processing, we do not distinguish between phase (2) and phase (3). Similar to the procedure in Case 1, R_3 $-$F $\rightarrow$ R_2 and R_4 $-$D $\rightarrow$ R_5 are profitable semijoins and are performed first. Then, it can be seen that attributes A and C are pure join attributes. Also, in light of Theorem 1 it can be shown that $|R_1 \text{ join } R_5| < |R_5|$ and $|R_1 \text{ join } R_2 \text{ join } R_5| < |R_2|$. Thus, instead of sending all relations toward the final site, we would like to use joins as reducers and perform $R_1 \Rightarrow R_5$ and then $R_5\prime \Rightarrow R_2$. Note that the semijoin $R_2 -$ B $\rightarrow R_1$ which incurs 900 units of transmission cost, though not profitable, is gainful with respect to the joins $R_1 \Rightarrow R_5$ and $R_5\prime \Rightarrow R_2$, and should be performed before the execution of $R_1 \Rightarrow R_5$ and $R_5\prime \Rightarrow R_2$. After $R_2 -$ B $\rightarrow R_1$, we have $|R_1\prime|$ =1190$\times$.0.75 =892.5, and the cost for $R_1\prime \Rightarrow R_5$ is thus 892.5$\times$.3 =2677.5. After $R_1\prime \Rightarrow R_5$ and the projection on attributes B, C, D of R_5, we get $|R_5\prime|$=1677.6 and the size of $R_5\prime$ is 1677.6$\times$.5 =8388 which is in turn the cost of $R_5\prime \Rightarrow R_2$. After the removal of attribute C from the resulting $R_2\prime$, we perform $R_2\prime \Rightarrow R_4$ and then $R_4\prime \Rightarrow R_3$, leading to

[7]The condition to determine if a semijoin is beneficial is formally formulated in Theorem 3 of Section 3.2.

the final result. In all, the transmission cost for each step is as follows. R_3−F $\rightarrow R_2$: 450, R_4−D $\rightarrow R_5$: 700, R_2−B $\rightarrow R_1$: 900, $R_1\prime \Rightarrow R_5$: 2678, $R_5\prime \Rightarrow R_2$: 8388, $R_2\prime \Rightarrow R_4$: 12 and $R_4\prime \Rightarrow R_3$: 62. Thus, the total transmission cost of Case 2 is 13190, which is significantly less than 35118 that is required in Case 1. The join and semijoin operations applied can be illustrated by the change of the query graph as shown in Fig. 6 where " $\Rightarrow$ " and " $\rightarrow$ " denote respectively a join and a semijoin operation.

3 Interleaving a Join Sequence with Beneficial Semijoins

As pointed out earlier, judiciously applying join operations as reducers can reduce the amount of data transmission required. Once a sequence of join reducers is determined, we need to identify the corresponding beneficial semijoins and determine the proper ordering of the join and semijoin reducers to achieve the most reduction in data transmission. Specifically, our approach to determine an effective sequence of join and semijoin reducers can be described in this section by the following five steps: (1) to obtain a join reducer sequence, (2) to map the join reducer sequence into a join sequence tree (in Section 3.1), (3) to derive the set of reducible relations for each semijoin (in Section 3.2), (4) to identify the beneficial semijoins based on the properties of beneficial semijoins developed, and (5) to determine the proper ordering in the combined reducer sequence (in Section 3.3).

3.1 Determining a join sequence tree

Note that there have been several methods proposed to obtain a join sequence. A polynomial time algorithm based on a mapping between a sequence of joins and a specific type of cut set to the query graph was developed in [9]. In addition, methods, such as dynamic programming [18] and A^* search [20], are alternative approaches to determine a join sequence. As can be seen later, our results in this paper to determine the beneficial semijoins do not rely upon any particular method to obtain the join sequence. In order not to distract the readers from the main theme of the paper, we do not include here the algorithms to obtain a join sequence. Interested readers are referred to [9] [18]. As pointed out earlier, the conventional approach of shipping all relations directly to the final site to perform the join operations after applying the semijoins is also one sort of join sequence, and thus a special case of our study. Such a join sequence tree is termed the *conventional join sequence tree* in what follows. Once a join sequence is determined, it can be mapped into its corresponding join sequence tree, which is defined as follows.

Definition 1: A join sequence tree is a rooted tree where each node denotes a relation and each edge implies a join between the two relations to which the edge is incident. The tree represents a sequence of join operations which are performed in such a way that each relation in a node is sent to its parent node in the tree for a join operation in the sense of bottom up.

More formally, we have the following lemma.

Lemma 1: A sequence of join operations for a query can be mapped into a join sequence tree.

Note that for the join sequence $R_1 \Rightarrow R_5$, $R_5\prime \Rightarrow R_2$, $R_4 \Rightarrow R_2\prime$ and $R_2\prime \Rightarrow R_3$, the corresponding join sequence tree is given in Fig. 2. It can be seen that the join sequence tree only implies a partial ordering on the join operations. In fact, the total communication required is immaterial to the order between the two joins $R_5 \Rightarrow R_2$ and $R_4 \Rightarrow R_2$ in Fig. 2, since there is no precedence imposed between the two joins. As it can be seen later, the concept of the join sequence tree will facilitate the derivation of theoretical results as well as the development of a heuristic algorithm for the approach of interleaving a join sequence with semijoins to produce an effective reducer sequence

in distributed query processing. Note that the join sequence tree corresponds to the sequence of join operations to be performed in a query graph, and should not be confused with the original query graph. For the join query tree in Fig. 7a with its profile shown in Table 2, it can be seen that there is no profitable semijoin in the given query graph. Thus, we can view this profile as if phases (1) and (2) of the query processing indicated in Section 1 had been performed. It can be verified that the cost of the join sequence, $R_2 \Rightarrow R_3$ and $R_3\prime \Rightarrow R_1$, is less than that of sending R_2 and R_3 directly to R_1. Thus $R_2 \Rightarrow R_3$ and $R_3\prime \Rightarrow R_1$ is the preferred join sequence. The corresponding join sequence tree is given in Fig. 7b which is different from Fig. 7a.

Recall that T_{R_i} is the subtree formed by R_i and its offspring in the join sequence tree, and $\mathrm{S}(T_{R_i})$ is the set of nodes in T_{R_i}. The *weight* of a relation R_i in the join sequence tree, denoted by $\mathrm{W}(R_i)$, is defined as the size of the resulting relation from joining all the relations in $\mathrm{S}(T_{R_i})$. That is, the weight of a relation R_i in a join sequence tree is equal to the cost of sending the relation resulting from joining those relations within T_{R_i} to the parent node of R_i. For the join sequence tree in Fig. 2, $\mathrm{W}(R_5) = w_{R_5\prime}|R_5\prime|$ and $\mathrm{W}(R_2) = w_{R_2\prime}|R_2\prime|$ where $R_5\prime$ is the relation resulting from joining R_1 and R_5, and $R_2\prime$ is the one from joining R_1, R_2, R_4 and R_5. For convenience the weight of the root of a join sequence tree, which corresponds to the final site, is defined to be zero. Also, to facilitate our study on the effect of semijoin operations, we define the *configuration* of a query, $J_Q(\mathrm{SMJ})$, to be the structure of the query and its profile associated after the set of semijoins SMJ has been performed. When it is necessary, we use $\mathrm{W}(R_i, J_Q(\mathrm{SMJ}))$, instead of $\mathrm{W}(R_i)$, to mean the weight of R_i after the semijoins in SMJ are performed.

Let $\mathrm{C}(J_Q(\mathrm{SMJ}))$ be the amount of data transmission required to complete the query according to a join sequence tree T in the configuration $J_Q(\mathrm{SMJ})$. $\mathrm{C}(J_Q(\mathrm{SMJ}))$ can be obtained by the following lemma.

Lemma 2: $\mathrm{C}(J_Q(\mathrm{SMJ})) = \sum_{R_i \in T} W(R_i, JQ(SMJ))$.

It can be seen that in the conventional join sequence tree as the one in Fig. 3, the weight of each relation is also the size of that relation. Thus, it can be verified that the cost of completing the phase (3) of the distributed query processing under the conventional approach is consistent with the result in Lemma 2.

3.2 Properties of beneficial semijoins

To study the properties of beneficial semijoins, we shall first investigate the effect of a semijoin on a join sequence tree. A relation is said to be *reducible* by a semijoin SJ_i if the weight of the relation in the join sequence tree is affected by the execution of the semijoin. Then, the set of reducible relations of a semijoin under a join sequence tree can be determined by the following theorem.

Theorem 2: Given a join sequence tree T, the set of reducible relations of a semijoin $R_i \rightarrow R_j$, denoted by $\mathrm{Rd}(R_i \rightarrow R_j)$, is $\mathrm{P}(R_i \vee R_j, R_j)$.

Proof: Recall that $R_i \vee R_j$ is the lowest common ancestor of R_i and R_j, and $\mathrm{P}(R_i \vee R_j, R_j)$ denotes the set of nodes on the path from $R_i \vee R_j$ to R_j excluding $R_i \vee R_j$. Note that the join operations are performed according to the corresponding join sequence tree in the sense of bottom up. After a semijoin is performed, the weights of those relations which are ancestors of the reducee of the semijoin will be reduced accordingly. However, for each relation R_k in the join sequence tree all the join operations in T_{R_k} have to be performed before R_k is sent to its parent node for another join operation. Thus, we know that R_i and R_j will be joined, together with other relations in $T_{R_i \vee R_j}$, in the site of $R_i \vee R_j$ before the resulting relation in $R_i \vee R_j$ can be sent to its parent node. This fact in turn implies that the effect of the semijoin $R_i \rightarrow R_j$ diminishes after the join in $R_i \vee R_j$. This theorem thus follows. **Q.E.D.**

For example, suppose Fig. 9 is the join sequence tree derived from Fig. 8, then Rd($R_1 \rightarrow R_4$)=$\{R_4, R_6, R_7\}$, Rd($R_4 \rightarrow R_3$)=$\{R_3\}$ and Rd($R_2 \rightarrow R_3$)=$\{R_3, R_6\}$. It is worth mentioning that in the conventional join sequence tree, the reducible set of a semijoin only consists of the reducee of that semijoin. This fact is described by the following corollary.

Corollary 2.1: Suppose $R_i \rightarrow R_j$ is a semijoin in a conventional join sequence tree where R_j is not the root, then Rd($R_i \rightarrow R_j$)=$\{R_j\}$.

For example, Rd($R_2 \rightarrow R_1$)=$\{R_1\}$ and Rd($R_5 \rightarrow R_4$)=$\{R_4\}$ in Fig. 3. Then, using the set of reducible relations the condition for a semijoin to be beneficial can be formally stated as follows.

Theorem 3: A semijoin SJ_k, $R_i - A \rightarrow R_j$ in the configuration J_Q(SMJ), is beneficial if and only if $w_A|R_i(A)| \leq (1-\rho_{i,a}) \sum_{R_p \in Rd(SJ_k)}$ W(R_p, J_Q(SMJ)).

Proof: Note that W(R_i, J_Q(SMJ)), the weight of R_i in J_Q(SMJ), is the size of the relation resulting from joining all the relations in T_{R_i}. From Theorems 1 and 2, we know that due to the execution of semijoin SJ_k the weight of a relation R_p in the set Rd($R_i \rightarrow R_j$) will be changed to $\rho_{i,a}$W(R_p, J_Q(SMJ)). Thus, it can be seen from Lemma 2 that due to the addition of SJ_k the total cost required to ship the relations in the set Rd($R_i \rightarrow R_j$) is changed from $\sum_{R_p \in Rd(SJ_k)}$ W(R_p, J_Q(SMJ)) to $w_A|R_i(A)| + \rho_{i,a} \sum_{R_p \in Rd(SJ_k)}$ W(R_p, J_Q(SMJ)), leading to this theorem. **Q.E.D.**

It can be seen from Theorem 3 that the condition for a semijoin to be beneficial in the conventional join sequence tree is similar to that of a profitable semijoin as applied in prior work [5], since a semijoin $R_i - A \rightarrow R_j$ in a conventional join sequence tree is beneficial if and only if $w_A|R_i(A)| \leq (1-\rho_{i,a})\, w_{R_j}|R_j|$. In addition, recall that $R_i \geq R_j$ means R_i is an ancestor of R_j in the join sequence tree. Note that Rd($R_j \rightarrow R_i$)=ϕ if $R_i \geq R_j$. Then, from this fact and Theorem 3 we have the following corollary which indicates that a node in a join sequence tree will not serve as a reducer for a semijoin to its ancestor. This agrees well with our intuition since the effect of such a semijoin will be offset by the subsequent join operations in T_{R_i}.

Corollary 3.1: Suppose R_i and R_j are two relations in a join sequence tree T and $R_i \geq R_j$. Then, $R_j \rightarrow R_i$ is not a beneficial semijoin for T.

Two semijoins are called *correlated* with each other if the condition for one to be beneficial depends on the execution of the other. Thus, using Theorem 3 we can determine by the following corollary if two semijoins are correlated with each other in a join sequence tree. Note that this concept will be used in the algorithm in Section 3.3 to determine the set of semijoins whose effect will be changed by the addition of a new semijoin.

Corollary 3.2: In a join sequence tree, two semijoins SJ_i and SJ_k are correlated with each other if and only if Rd(SJ_i) $\cap$ Rd(SJ_k) $\neq \phi$.

For example, for the join sequence tree in Fig. 9, $R_3 \rightarrow R_4$ and $R_2 \rightarrow R_6$ are not correlated since Rd($R_3 \rightarrow R_4$) $\cap$ Rd($R_2 \rightarrow R_6$) =ϕ. However, $R_3 \rightarrow R_4$ and $R_2 \rightarrow R_7$ are correlated since Rd($R_3 \rightarrow R_4$) $\cap$ Rd($R_2 \rightarrow R_7$) =$\{R_7\}$. It is interesting to see that under the conventional approach in which only profitable semijoins are concerned, two semijoins are correlated with each other only when they have the same reducee. This fact can also be described by the corollary below which follows directly from Corollaries 2.1 and 3.2.

Corollary 3.3: In a conventional join sequence tree two semijoins are correlated if and only if they have the same reducee.

3.3 Algorithm to determine beneficial semijoins for a join sequence tree

In light of the properties of beneficial semijoins derived in Section 3.2, we can develop a heuristic algorithm to determine the beneficial semijoins for a join sequence tree as given in algorithm G below. Note that the condition for a semijoin on J_Q(SMJ) to be beneficial can be determined by Theorem

3. To simplify the presentation of the algorithm, we use a Boolean function B(SJ_i,J_Q(SMJ)) to denote the outcome of such a condition, i.e., B(SJ_i,J_Q(SMJ)) is "true" (respectively, "false") if SJ_i is (respectively, is not) beneficial in the configuration J_Q(SMJ).

We use SM_T to denote the set of possible semijoins in the original query graph, and SMJ to mean the set of beneficial semijoins identified thus far. Define the *cumulative benefit* of a semijoin SJ_k, $R_i - A \rightarrow R_j$, denoted by CB(SJ_k), as the amount of reduction if it is applied individually prior to the execution of a given join sequence. i.e., CB(SJ_k)= $(1-\rho_{i,a}) \sum_{R_p \in Rd(SJ_k)}$ W(R_p, $J_Q(\phi)$). Note that the cumulative benefit of a semijoin is in fact the same as the benefit of a semijoin as in most prior work [27] when the corresponding join sequence tree is the conventional join sequence tree. Clearly, the cumulative benefit can be used as a heuristic to determine the order of semijoins to be evaluated so that semijoins with larger cumulative benefits can be considered first. Thus, this algorithm checks each semijoin in SM_T, according to a descending order of their cumulative benefits, to see if that semijoin should be included into SMJ according to the configuration J_Q(SMJ) in line 6. If "yes" in line 6, we shall include SJ_i into SMJ (line 8) and then check if any semijoin previously included in SMJ should be removed due to the addition of SJ_i (line 9 to line 11). Note that in light of Corollary 3.2, only those semijoins that are correlated with SJ_i, denoted by SM_B, have to be re-evaluated, thus reducing the computational cost required. On the other hand, if "no" in line 6, we shall check if the semijoin SJ_i is beneficial in the original configuration $J_Q(\phi)$ (line 14). If "yes", we know that the semijoin SJ_i is not beneficial due to the existence of some other semijoins in SM_B which are correlated with SJ_i. We thus check in line 16 to line 18 to determine if it is worth while replacing those semijoins in SM_B with SJ_i. Note that C(SM_B) is the cost of executing semijoins in SM_B and C($\{SJ_i\}$) is that of executing SJ_i. This algorithm can be formally described as follows.

Algorithm G : Determine beneficial semijoins for a join sequence.

1. Determine the set of possible semijoins SM_T from the query graph such that the condition in Corollary 3.1 is satisfied.
2. Sort the semijoins in SM_T in a descending order of their cumulative benefits, i.e., CB(SJ_i)≥CB(SJ_j) if i≤j.
3. SMJ:= ϕ;
4. **for** i=1, $|SM_T|$ **do**
5. **begin**
6. **if** B(SJ_i, J_Q(SMJ)) **then**
7. **begin**
8. SMJ:= SMJ ∪ $\{SJ_i\}$;
9. SM_B :=$\{SJ_q|$ Rd(SJ_q)∩Rd(SJ_i) ≠ ϕ and SJ_q ∈ SMJ$\}$;
10. **for** $SJ_k \in SM_B$ **do**
11. **if** ¬B(SJ_k, J_Q(SMJ − $\{SJ_k\}$)) **then** SMJ:= SMJ−$\{SJ_k\}$;
12. **end**
13. **else**
14. **if** B(SJ_i,$J_Q(\phi)$) **then**
15. **begin**
16. SM_B :=$\{SJ_q|$ Rd(SJ_q) ∩ Rd(SJ_i) ≠ ϕ and SJ_q ∈ SMJ$\}$;
17. **if** C(J_Q(SMJ))+C(SM_B) > C(J_Q(SMJ $-SM_B \cup \{SJ_i\}$)) +C($\{SJ_i\}$)
18. **then** SMJ:= SMJ−$SM_B \cup \{SJ_i\}$;
19. **end**
20. **end**

Note that for a query of n relations, the complexity of determining B(SJ_i, J_Q(SMJ)) is O(n). From this fact and the operations in lines 10 and 11, it can be shown that the worst case complexity of algorithm G is $O(|SM_T|^2 n)$. Moreover, in light of Corollary 3.3, it can be seen that when the join sequence tree is the conventional join sequence tree, algorithm G will degenerate into the one of determining a set of profitable semijoins, a version similar to the algorithm in [3].

After the beneficial semijoins are identified by the above algorithm, these semijoins can be inserted into the join sequence according to the procedure given below. In this procedure, we perform the operation, a join or a semijoin, from the leaf nodes of the join sequence tree. In Step 1, a join operation associated with a leaf node is performed as long as the leaf node is neither a reducer nor a reducee of a semijoin operation in SMJ. Step 1 is repeated until there is no such a join available. In Step 2 and Step 3, we then perform proper semijoins to enable the execution of Step 1 while minimizing the cost required for semijoins.

Procedure P : Determine the order of join and semijoin reducers.

Step 1: In the join sequence tree, perform join operations associated with leaf nodes which are neither reducers nor reducees of the semijoins in SMJ.

Update the join sequence tree by merging the leaf node to its parent node after each join operation is performed.

Repeat Step 1 until there is no such a join available.

Step 2: If there is a semijoin SJ_i in SMJ, of which the reducer is a leaf node of the join sequence tree, then perform SJ_i, remove SJ_i from SMJ, and go to Step 1.

Otherwise, go to Step 3.

Step 3: /* All the leaf nodes in the join sequence tree are reducees of the remaining semijoins in SMJ. */

Choose a semijoin SJ_k with the smallest cost from SMJ.

Perform SJ_k and remove it from SMJ.

Go to Step 1.

It is worth mentioning that the execution of join or semijoin operation for a relation will reduce the cardinalities of attributes in that relation. This is also true for those attributes which are neither join nor semijoin attributes in that relation. This fact can be verified by Eq.(1). This is the very reason that in procedure P while exploiting each semijoin, we do not execute a semijoin until it is necessary so as to reduce the cost of data transmission required for the semijoin operation. The operations in algorithm G and procedure P can be illustrated by the examples in the following section.

4 Remarks and Examples

To show the execution of algorithm G, consider the query graph in Fig. 8 with the join sequence tree in Fig. 9. Suppose that each edge is associated with one attribute[8]. It can be seen from Fig. 8 that there are 16 possible semijoins for the query. The selectivities and costs of these semijoins are given in Table 3. In light of the join sequence tree in Fig. 9, it can be verified using Corollary 3.1

[8]This assumption is not essential but will simplify our example.

that only 10 semijoins are potentially beneficial to be included into SM_T. In Table 3, the column "in SM_T" identifies the 10 semijoins. The 10 semijoins are illustrated in Fig. 10. Also, we assume that the sizes and weights of relations in the join sequence tree are those in Table 4. Recall that the weight of a relation is the size of the resultant relation from joining that relation and all its offspring, which can be determined by Theorem 1. It can be verified that the cost of executing the join sequence without applying semijoins is $\sum_{i=1}^{7} \mathrm{W}(R_i)$= 10045.

Using the profile in Tables 3 and 4, the beneficial semijoins can be identified from SM_T by algorithm G. Note that the cumulative benefit of $R_2 \rightarrow R_6$ is determined by 820*0.2−153= 11 and that of $R_1 \rightarrow R_4$ is (2400+820+2125)*0.93 −276=98. Similarly, the cumulative benefit of each semijoin can be obtained and shown in column "CB(SJ_i)" of Table 3. Then, the semijoins are evaluated by algorithm G according to their order in Table 5. It can be seen that the first two semijoins $R_3 \rightarrow R_4$ and $R_6 \rightarrow R_3$ are beneficial and thus included into SMJ. Note that due to the addition of the two semijoins, the weights of R_3, R_4 and R_7 have to be modified accordingly, i.e., W(R_3), W(R_4) and W(R_7) become 1440, 1680 and 1488 respectively. This accounts for the reason that $R_1 \rightarrow R_4$ is not included into SMJ. Following the same procedure, semijoins $R_4 \rightarrow R_1$, $R_2 \rightarrow R_6$ and $R_7 \rightarrow R_5$ will also be included into SMJ by the operations of algorithm G as indicated in Table 5, resulting in five beneficial semijoins for the query in Fig. 8.

It is worth mentioning that a different order of semijoins evaluated in line 4 may result in a different set of beneficial semijoins. To illustrate this fact and show more insights to the operations in algorithm G, consider the case that the semijoins in SM_T are evaluated according to the order in Table 6. In such a case, it can be seen that the first three semijoins $R_2 \rightarrow R_6$, $R_2 \rightarrow R_1$ and $R_6 \rightarrow R_7$ are beneficial and thus included into SMJ first. Then, semijoins $R_6 \rightarrow R_3$, $R_7 \rightarrow R_5$ and $R_4 \rightarrow R_1$ will also be included into SMJ by the operations of algorithm G. Notice that, as indicated in Table 6, $R_4 \rightarrow R_1$ was not beneficial at first. However, it becomes beneficial after the removal of $R_2 \rightarrow R_1$. This is a result of the operations in line 16 to line 18 of algorithm G. Also note that after the operations in line 9 to line 11 of algorithm G, the addition of $R_1 \rightarrow R_4$ whose reducible set is $\{R_4, R_6, R_7\}$ will cause the semijoins $R_2 \rightarrow R_6$ and $R_6 \rightarrow R_7$ to become no longer beneficial and thus be removed from SMJ. By the same reason, $R_1 \rightarrow R_4$ is removed later due to the addition of $R_3 \rightarrow R_4$, resulting to the four beneficial semijoins as indicated in Table 6. It can be seen that semijoin $R_2 \rightarrow R_6$, which should be included into the final SM_T, is absent in the resulting set of Table 6, since it was removed due to the addition of $R_1 \rightarrow R_4$ which is, however, deleted later by the addition of $R_3 \rightarrow R_4$. This is the very reason we evaluate in algorithm G the semijoins according to the descending order of their cumulative benefits so that more beneficial semijoins can be included first, avoiding unnecessary or even incorrect addition/deletion of semijoins as shown in Table 6.

After the execution of algorithm G as in Table 5, the semijoins kept in SMJ are $R_2 \rightarrow R_6$, $R_6 \rightarrow R_3$, $R_7 \rightarrow R_5$, $R_4 \rightarrow R_1$ and $R_3 \rightarrow R_4$. An illustration of these beneficial semijoins for the join sequence tree can be found in Fig. 11. The final join and semijoin reducers, $R_7 \rightarrow R_5$, $R_5 \Rightarrow R_7$, $R_4 \rightarrow R_1$, $R_1 \Rightarrow R_2$, $R_3 \rightarrow R_4$, $R_4 \Rightarrow R_7$, $R_7 \Rightarrow R_6$, $R_6 \rightarrow R_3$, $R_3 \Rightarrow R_6$, $R_2 \rightarrow R_6$, and $R_6 \Rightarrow R_2$, can then be determined from procedure P in Section 3.3. It can also be obtained that the total cost of data transmission is 200+490 +370+1760 +206+1680 +1488+200 +1440+ 153+656= 8643, significantly less than 10045 that is previously required when semijoins have not been inserted into the join sequence. It is worth mentioning that if we only apply semijoins as reducers and evaluate the semijoins based on the algorithm in [3] according to their order in Table 3, semijoins $R_2 \rightarrow R_6$, $R_2 \rightarrow R_1$, $R_6 \rightarrow R_3$, $R_7 \rightarrow R_5$ and $R_3 \rightarrow R_4$ will be identified as profitable semijoins which are not the same as those obtained by algorithm G. The total cost of data transmission required for this approach is 10901, including the cost of performing the 5 profitable semijoins and that of sending all the remaining relations to the final site. It can be seen from the above example and the one in Section 2.2.3 that

the approach of combining join and semijoin reducers is effective in reducing the amount of data transmission required.

5 Conclusion

In this paper, we studied the problem of combining join and semijoin reducers for distributed query processing. An approach of interleaving a join sequence with beneficial semijoins was proposed. A join sequence was mapped into a join sequence tree first. The join sequence tree provides an efficient way to identify for each semijoin its correlated semijoins as well as its reducible relations under the join sequence. In light of these properties, we developed an algorithm to determine an effective sequence of join and semijoin reducers. Examples were also given to illustrate our results. Specifically, our approach to determine an effective sequence of join and semijoin reducers consists of the following five steps: (1) to obtain a join reducer sequence (e.g. based on the algorithm in [9]), (2) to map the join reducer sequence into a join sequence tree, (3) to derive the set of reducible relations for each semijoin (see Theorem 2), (4) to identify the beneficial semijoins based on the properties of beneficial semijoins developed in Section 3.2 (see algorithm G), and (5) to determine the proper ordering in the combined reducer sequence (see Procedure P). Our results showed the advantage of using a combination of joins and semijoins as reducers for distributed query processing.

ACKNOWLEDGEMENT

The authors would like to thank J.-C. Chen at IBM for her technical assistance in preparing this paper.

REFERENCES

[1] P. M. G. Apers, A. R. Hevner and S. B. Yao, "Optimization Algorithms for Distributed Queries," *IEEE Trans. on Software Eng.,* vol. SE-9, no. 1, pp. 57-68, Jan. 1983.

[2] P. A. Bernstein and D.-M. W Chiu "Using Semi-Joins to Solve Relational Queries," *Journal of ACM,* vol. 28, no. 1, pp. 25-40, Jan. 1981.

[3] P. A. Bernstein, N. Goodman, E. Wong, C. Reeve and J. B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Syst.,* vol. 6, no. 4, pp. 602-625, Dec. 1981.

[4] P. A. Black and W. S. Luk, "A New Heuristic for Generating Semi-Join Programs for Distributed Query Processing," *Proceedings of IEEE COMPSAC,* pp. 581-588, 1982.

[5] S. Ceri and G. Pelagatti, *Distributed Databases Principle and Systems,* NY.: McGraw-Hill, 1985.

[6] A. L. P. Chen and V. O. K. Li, "Improvement Algorithms for Semijoin Query Processing Programs in Distributed Database Systems," *IEEE Trans. on Comput.,* vol. C-33, no. 11, pp. 959-967, Nov. 1984.

[7] A. L. P. Chen and O. K. Li, "Optimizing Star Queries in a Distributed Database System," *Proceedings of the 10th Int'l Conf. on Very Large Data Bases,* pp. 429-438, Aug. 1984.

[8] M.-S. Chen and P. S. Yu, "Using Combination of Join and Semijoins Operations for Distributed Query Processing," *Proceedings of the 10th Int'l Conf. on Distributed Computing Systems,* pp. 328-335, May 1990.

[9] M.-S. Chen and P. S. Yu, "Using Join Operations as Reducers in Distributed Query Processing," *Proceedings of the 2nd Intern'l Symp. on Databases in Parallel and Distributed Systems*, July 1990. Also, IBM Research Report RC 15107, Nov. 1989.

[10] D.-M. Chiu, P. A. Bernstein and Y.-C. Ho, "Optimizing Chain Queries in a Distributed Database System," *SIAM Journal on Computing,* vol. 13, no. 1, pp. 116-134, Feb. 1984.

[11] W. W. Chu and P. Hurley, "Optimal Query Processing for Distributed Database Systems," *IEEE Trans. on Comput.,* vol. C-91, no. 9, pp. 135-150, Sep. 1982.

[12] N. Goodman and O. Shmueli, "The Tree Property is Fundamental for Query Processing," *Proceedings ACM Symp. on Principles of Database Systems,* pp. 40-48, 1982.

[13] F. Harary, *Graph Theory,* Mass.: Addison-Wesley, 1969.

[14] A. R. Hevner, "The Optimization of Query Processing on Distributed Database Systems," Ph.D. Dissertation, Purdue University, 1979.

[15] A. R. Hevner and S. B. Yao, "Query Processing in Distributed Database Systems," *IEEE Trans. on Software Eng.,* vol. SE-5, no. 5, pp. 177-187, May 1979.

[16] Y. Kambayashi, M. Yoshikawa and S. Yajima, "Query Processing for Distributed Databases Using Generalized Semi-Joins," *ACM Proceedings of SIGMOD,* pp. 151-160, 1982.

[17] H. Kang and N. Roussopoulos, "Combining Joins and Semijoins in Distributed Query Processing," Univ. Maryland, College Park, Tech. Rep. CS-TR-1794, 1987.

[18] S. Lafortune and E. Wong, "A State Transition Model for Distributed Query Processing," *ACM Trans. on Database Systems,* vol. 11, no. 3, pp. 294-322, Sep. 1986.

[19] G. M. Lohman, C. Mohan, L. M. Hass, B. G. Lindsay, P. G. Selinger, P. F. Wilms and D. Daniels, "Query Processing in R^*," Res. Rep. RJ 4272, IBM Research Laboratory, San Jose, CA., April 1984.

[20] N. J. Nilsson, *Principles of Artificial Intelligence.* Springer-Verlag, 1982.

[21] S. Pramanik and D. Vineyard, "Optimizing Join Queries in Distributed Databases," *IEEE Trans. on Software Eng.,* vol. SE-14, no. 9, pp. 1319-1326, Sep. 1988.

[22] A. Segev, "Global Heuristic for Distributed Query Optimization," *Proceedings of IEEE INFOCOM,* pp. 388-394, 1986.

[23] C. Wang, "The Complexity of Processing Tree Queries in Distributed Databases," *Proceedings of the 2nd IEEE Symp. on Parallel and Distributed Processing,* pp. 604-611, December 1990.

[24] S. B. Yao, "Approximating Block Access in Database Organizations," *Comm. of ACM,* vol. 20, pp. 260-261, Apr. 1977.

[25] H. Yoo and S. Lafortune, "An Intelligent Search Method for Query Optimization by Semijoins," *IEEE Trans. on Knowledge and Data Eng.,* vol. 1, no. 2, pp. 226-237, June 1989.

[26] C. T. Yu, Z. Ozsoyoglu and K. Lam, "Optimization of Distributed Tree Queries," *J. Comput. Syst. Sci.,* vol. 29, no. 3, pp. 409-445, Dec. 1984.

[27] C. T. Yu and C. C. Chang, "Distributed Query Processing," *ACM Computing Surveys,* vol. 16, no. 4, pp. 399-433, Dec. 1984.