On Cost-effectiveness of a Semijoin in Distributed Query Processing

by

Hyunchul Kang
Nick Roussopoulos
On Cost-effectiveness of a Semijoin in Distributed Query Processing

by

Hyunchul Kang
Nick Roussopoulos
January 1987

On Cost-effectiveness of a Semijoin in Distributed Query Processing

By

Hyunchul Kang
Nick Roussopoulos

Database Systems Group
Department of Computer Science
University of Maryland
College Park, MD 20742

This research was sponsored partially by National Science Foundation under the grant CDR-85-00108.
Abstract

The cost-effective reduction of relations by semijoins is the basis of the heuristic approach to distributed query processing. The cost-effectiveness of a semijoin was simply determined in the literature assuming that the local processing cost is negligible compared to the data transmission cost in distributed query processing. However recently, questions have been raised about the validity of the assumption, and some experimental works revealed that the local processing cost is also significant in distributed query processing. In this paper, we are concerned with the cost-effectiveness of a semijoin considering the local processing cost as well as the data transmission cost. To measure the effectiveness of a semijoin in terms of the local processing cost, we introduce the join sequence in which the relations are joined at the result site to answer the query. A dynamic programming algorithm is developed to generate the optimal join sequence for a given query. A simple heuristic algorithm is also developed to generate a join sequence for a given query.
1. Introduction

In distributed database systems, processing a query involves data transmission among different sites of the computer network. The cost of processing a query consists of the data transmission cost as well as the local processing cost.

The operation which involves data transmission in distributed query processing is the join of relations stored at different sites. The operations like selection and projection are performed at each site storing relations referenced in the query before joins are processed. The optimization of the selection and projection operations is well understood from the query optimization in the centralized database environments [Wong and Youssefi 76][Selinger et al 79][Yao 79]. Thus, in distributed query processing, it is assumed that the query consists of only join operations.

In the literature, a popular approach to distributed query processing is to process a query by the following two phases:

(1) reducing phase: executes a semijoin program which is a sequence of semijoins.
For two relations to be joined, a semijoin [Bernstein and Chiu 81] sends the joining attribute values of one relation to the site of the other and reduces the other relation by eliminating tuples which are not joinable.

(2) joining phase: sends all the relations, possibly reduced in the reducing phase, to the result site where the query result is requested and joins them to answer the query.

The problem of generating semijoin programs has been a great interest to a number of researchers [Wong 77][Bernstein et al 81][Black and Luk 82][Hevner and Yao
It was proved that the problem of generating the optimal semijoin program is NP-hard [Hevner 79]. Therefore, the computationally feasible algorithms to generate semijoin programs depend on heuristics.

One of the representative heuristics is that the semijoin included in the semijoin program should be cost-effective. A semijoin is defined as cost-effective if its cost is less than its benefit. The cost of a semijoin consists of the costs of projecting and sending the joining attribute values of one relation to the site of the other and of reducing the other relation. The benefit consists of the cost reductions in sending and processing the reduced relation due to its reduction. It is always profitable to execute a cost-effective semijoin. To see this point, for two relations $R_1$ and $R_2$ at different sites, consider a semijoin from $R_1$ to $R_2$ denoted as $R_1 \rightarrow R_2$ which sends the joining attribute values of $R_1$ to the site of $R_2$ to reduce $R_2$ to $R_2'$. Suppose $R_1 \rightarrow R_2$ is cost-effective. Let the cost of sending and processing $R_2$ and $R_2'$ be $c_2$ and $c_2'$, respectively. Then, the benefit of $R_1 \rightarrow R_2$ is $b = c_2 - c_2'$, and since $R_1 \rightarrow R_2$ is cost-effective, its cost $c < b$. Thus, we have $c + c_2' < c_2$, which implies that it is profitable to execute $R_1 \rightarrow R_2$.

In this paper, we are concerned with the cost-effectiveness of a semijoin in distributed query processing. In the literature, algorithms proposed to generate semijoin programs assumed that the local processing cost is negligible compared to the data transmission cost. Thus, the cost-effectiveness of a semijoin could be simply determined. The cost of $R_1 \rightarrow R_2$ was the transmission cost for the joining attribute values of $R_1$, and the benefit was the difference in the transmission costs for $R_2$ and $R_2'$.

---

1 Given a query, a semijoin program $P$ is optimal if the cost to execute $P$ in the reducing phase and to send and join the relations at the result site in the joining phase is minimum among all the possible semijoin programs.
However recently, questions have been raised about the validity of the assumption that the local processing cost is negligible compared to the data transmission cost. Some experimental works revealed that the local processing cost is also significant in query processing even in a geographically distributed environment [Lu and Carey 85][Mackert and Lohman 86]. In this paper, we take the position that the local processing cost is also significant in distributed query processing. In such distributed environments, the cost and benefit of a semijoin should be recomputed by considering the local processing cost as well as the data transmission cost.

The cost of a semijoin is a mere sum of the costs of the local processing component and of the data transmission component in executing the semijoin. However, the benefit computation is more complicated for which we introduce the *join sequence*. In the joining phase of processing a query, all the relations possibly reduced in the reducing phase are sent to the result site to be joined. We call the sequence in which the relations are joined at the result site in the joining phase as the join sequence. The join sequence is used to measure the local processing cost of joining the relations at the result site in the joining phase. Given a query, to generate the optimal join sequence we develop a dynamic programming algorithm.\(^2\) Since this algorithm runs in an exponential time, we also develop a simple heuristic algorithm which generates a join sequence in a polynomial time.

The rest of this paper is organized as follows. In section 2, a cost model of distributed query processing is presented. Section 3 formulates the cost of a semijoin and introduces the join sequence to formulate the benefit of a semijoin. In section 4, we develop a dynamic programming algorithm which generates the optimal join sequence for

---

\(^2\) The optimality of the join sequence will be defined in section 3.
a given query. In section 5, we develop a heuristic algorithm to generate a join sequence for a given query. Concluding remarks are in section 6.

2. Cost Model

In this section, a cost model of distributed query processing is presented. The cost of processing a query consists of the data transmission cost and the local processing cost.

The data transmission cost of sending relation \( R \) from one site to another in the computer network is measured by the following formula.

\[
T(R) = t_0 + t_1 s(R)
\]

where \( t_0 \) and \( t_1 \) are constants and \( s(R) \) is the size of \( R \).\(^3\) This formula has been widely used by many researchers in the literature.

On the other hand, there is no representative formula around to measure the local processing cost in distributed query processing. This is because the local processing cost has been assumed to be negligible compared to the data transmission cost in the literature.

For two relations \( R_1 \) and \( R_2 \) to be joined, we use the following formula to measure the local processing cost of computing this join.

\[
l_j(R_1, R_2) = l_0 s(R_1) s(R_2)
\]

where \( l_0 \) is a constant.

To compute a semijoin, the joining attribute values of one relation are projected first. We use the following formula to measure the local processing cost of projecting the joining attribute values from relation \( R \).

\[
l_p(R) = l_1 s(R)
\]

\(^3\) \( t_0 \) is the communication setup time and \( t_1 \) is the data transmission rate.
where \( l_1 \) is a constant.

The above formulas for \( l_j \) and \( l_p \) assume that there is no access path such as the secondary index available to speed up the joins or projections. This is justified by the observations that in distributed query processing,

(1) Most of the relations to be processed are intermediate result relations generated by semijoins and joins leading to the result of the query. The intermediate result relation is not supported by access paths unless some are created dynamically.

(2) The operations like selection and projection are performed before semijoins and joins are computed. Thus, most of the relations to be semijoin and joined are also intermediate result relations.

As for the relative importance between the data transmission cost and the local processing cost in processing a query, a constant \( w \) is used such that a unit of the data transmission cost corresponds to \( w \) units of the local processing cost. Let

\[
L_j(R_1, R_2) = w \ l_j(R_1, R_2) = w \ l_0 \ s(R_1) \ s(R_2)
\]

\[
L_p(R) = w \ l_p(R) = w \ l_1 \ s(R)
\]

3. Cost and Benefit of Semijoin

3.1. Cost of Semijoin

For two relations \( R_1 \) and \( R_2 \) at different sites, consider a semijoin from \( R_1 \) to \( R_2 \) on attribute \( A \) denoted as \( R_1 \rightarrow A \rightarrow R_2 \). This semijoin projects attribute \( A \) from \( R_1 \), sends \( R_1[A] \) to the site of \( R_2 \) and reduces \( R_2 \) by joining \( R_1[A] \) and \( R_2 \). Thus, according to the cost model presented in the previous section, the cost of \( R_1 \rightarrow A \rightarrow R_2 \) is

\[
L_p(R_1) + T(R_1[A]) + L_j(R_1[A], R_2)
\]
3.2. Benefit of Semijoin

3.2.1. Join Sequence

For $n \geq 2$ relations $R_1, \ldots, R_n$ at the same site, there can be more than one different sequence in which they are joined to generate the join result $\text{join}(R_1, \ldots, R_n)$. We call the sequence in which relations are joined as the join sequence.

For example, for $n = 3$ relations $R_1, R_2$, and $R_3$, assuming that any pair of relations is joinable, there are 6 different join sequences to generate $\text{join}(R_1,R_2,R_3)$, as depicted in Figure 1. If $R_2$ and $R_3$ are not joinable, then the sequences in Figure 1(b),(d) and (e) are not applicable, reducing the number of different join sequences to 3. For $n$ relations, assuming that any pair of relations is joinable, as $n$ grows, the number of different join sequences grows exponentially.

Formally, the join sequence is a partial ordering of unordered pairs of relations $\langle R_x, R_y \rangle$ where $R_x$ ($R_y$) is either some $R_i$, $1 \leq i \leq n$ or some intermediate join result $\text{join}(R_{i_1}, \ldots, R_{i_k})$, $1 \leq i_k \leq n$, $i = 1, \ldots, j$, $j \geq 2$ such that each pair $\langle R_x, R_y \rangle$ in the ordering represents the join of $R_x$ and $R_y$, and the joins in the ordering are to be computed in the specified order.

For example, the join sequence in Figure 1(a) is

$\langle R_1, R_2 \rangle$ followed by $\langle \text{join}(R_1, R_2), R_3 \rangle$

and the join sequence in Figure 1(d) is

$\langle R_1, R_2 \rangle$ and $\langle R_2, R_3 \rangle$ followed by $\langle \text{join}(R_1, R_2), \text{join}(R_2, R_3) \rangle$

where there is no precedence between $\langle R_1, R_2 \rangle$ and $\langle R_2, R_3 \rangle$.

The cost of a join sequence is defined as the sum of costs to compute joins in the sequence. For example, the cost of the join sequence in Figure 1(a) is
\[ L_j(R_1, R_2) + L_j(\text{join}(R_1, R_2), R_3) \]

For a query joining the relations stored at different sites, the join sequence is the sequence in which the relations are joined at the result site in the joining phase. The optimal join sequence for a given query is defined as the join sequence with the minimum cost among all the possible join sequences for the query.

### 3.2.2. Computation of Benefit

For two relations \( R_1 \) and \( R_2 \) at different sites, consider a semijoin \( R_1 \rightarrow R_2 \) which reduces \( R_2 \) to \( R_2' \). The benefit of this semijoin is defined as the cost difference between before the semijoin and after the semijoin in sending \( R_2 \) to the result site and in joining \( R_2 \) with other relations referenced in the query. The difference in the data transmission cost is

\[ T(R_2) - T(R_2') \]

as was computed in the literature. To compute the difference in the local processing cost to join \( R_2 \) with other relations, we use a join sequence for the query. For join sequence \( S \), let \( L_{js}(S) \) be the cost of \( S \) and let \( L_{js}(S(R_2' / R_2)) \) be the cost of \( S \) with all occurrences of \( R_2 \) replaced by \( R_2' \). Then, the difference is

\[ L_{js}(S) - L_{js}(S(R_2' / R_2)) \]

Thus, the benefit of \( R_1 \rightarrow R_2 \) with respect to join sequence \( S \) is

\[ | T(R_2) + L_{js}(S) | - | T(R_2') + L_{js}(S(R_2' / R_2)) | \]

We note that the benefit of a semijoin is computed with respect to a particular join sequence for the query. Therefore given a query, before the semijoin program is generated, join sequence \( S \) is generated by some algorithm to be used for the computation of the benefit of semijoins. During the generation of the semijoin program, each time a semijoin, say \( R_1 \rightarrow R_2 \), is selected to be included in the semijoin program, the current
join sequence \( S \) is replaced by \( S(R_2'/R_2) \).

4. Dynamic Programming Algorithm

4.1. Algorithm OJS

In this section, we develop an algorithm to generate the optimal join sequence for a given query.

For a set of relations \( R = \{ R_1, \ldots, R_n \} \), let \( \text{join}(R) \) be

\[
\begin{cases}
  R_1 & \text{if } n = 1, \text{ that is, } R \text{ is a singleton set } \{ R_1 \} \\
  \text{join}(R_1, \ldots, R_n) & \text{if } n \geq 2
\end{cases}
\]

Given a set of relations \( R = \{ R_1, \ldots, R_n \} \), consider a query referencing the relations in \( R \) to generate \( \text{join}(R) \). The join result \( \text{join}(R) \) is generated by joining two relations \( R_l \) and \( R_r \) where \( R_l \) (\( R_r \)) is either some \( R_i, 1 \leq i \leq n \) or some intermediate join result \( \text{join}(R_{i_1}, \ldots, R_{i_j}), 1 \leq i_k \leq n, k = 1, \ldots, j, j \geq 2 \). \( R_l \) (\( R_r \)) corresponds to a proper subset \( R_L \) (\( R_R \)) of \( R \) such that

\[
R_l = \text{join}(R_L) \quad \quad R_r = \text{join}(R_R)
\]

That is, \( R_l \) (\( R_r \)) is the result of the subquery referencing the relations in \( R_L \) (\( R_R \)). If \( R_L \) (\( R_R \)) is a singleton set, the relation in \( R_L \) (\( R_R \)) vacuously forms a subquery. Otherwise, in order for the relations in \( R_L \) (\( R_R \)) to form a subquery, the relations should be joinable to generate the join result \( \text{join}(R_L)(\text{join}(R_R)) \). We denote this condition as

JOINABLE(R_L)

JOINABLE(R_R)

There are some more conditions that \( R_L \) and \( R_R \) should satisfy:

\[ R_L \neq R_R \text{ but } R_L \cup R_R = R \]

Otherwise, we could not generate the correct \( \text{join}(R) \).
Otherwise, we could not generate the correct \( \text{join}(R) \).

Given a set of relations \( R \), let \( \text{COST}(R) \) be the cost of the optimal join sequence for the query referencing the relations in \( R \) to generate \( \text{join}(R) \). \( \text{COST}(R) \) is 0 if \( R \) is a singleton set. For this query, let \( LR(R) \) be the set of all the possible \( <R_L, R_R> \) pairs such that

\[
R_L \subset R, \ R_R \subset R, \\
R_L \neq \emptyset, \ R_R \neq \emptyset, \\
\text{JOINABLE}(R_L) \text{ and JOINABLE}(R_R), \\
R_L \neq R_R, \ R_L \cup R_R = R
\]

Then, we have the following recurrence relation,

\[
\text{COST}(R) = \min_{<R_L, R_R> \in LR(R)} \left( \text{COST}(R_L) + \text{COST}(R_R) + L_f(\text{join}(R_L), \text{join}(R_R)) - \text{COMMON}(R_L, R_R) \right)
\]

This relation is derived from the principle of optimality stating that the subsequence of the optimal join sequence for a query should be the optimal join sequence for the corresponding subquery. The term \( \text{COMMON}(R_L, R_R) \) is the cost of the join sequences common in the join subsequences for \( R_L \) and \( R_R \). A dynamic programming algorithm can solve this relation using the knowledge \( \text{COST}(R) = 0 \) when \( R \) is a singleton set and \( \text{COST}(R) = L_f(R_1, R_2) \) when \( R = \{ R_1, R_2 \} \).

For a set of relations \( R \) referenced in a query, algorithm OJS generates the optimal join sequence for the query as follows. For each subset \( R^* \) of \( R \), \( \text{COST}(R^*) \) is computed starting from the subsets with one relation to the subset equal to \( R \). While computing \( \text{COST}(R^*) \), if \( |R^*| \geq 3 \), the pair \( <R^*_{L}, R^*_{R}> \in LR(R^*) \) which minimizes the above recurrence relation for \( \text{COST}(R^*) \) is recorded as \( L(R^*) = R^*_{L} \) and \( R(R^*) = R^*_{R} \). The optimal join sequence can be generated recursively by subalgorithm \( SEQ \) using these
\[ L(R^*) \text{ and } R(R^*) \] for each subset \( R^* \) of \( R \).

**Algorithm OJS**

input: a set of \( n \) relations \( R = \{ R_1, \ldots, R_n \} \) referenced in a query
output: the optimal join sequence \( S \)

begin

for \( i = 1 \) to \( n \) \( \text{COST}(\{ R_i \}) := 0 \)

for \( i = 1 \) to \( \left( \frac{n}{2} \right) \) do

let \( R_i \) be the \( i \)-th subset of \( R \) such that \( R_i = \{ R_{j_1}, R_{j_2} \} \)

if \( \text{JOINABLE}(R_i) \) then \( \text{COST}(R_i) := L_j(R_{j_1}, R_{j_2}) \)

end

for \( i = 3 \) to \( n \) do

for \( j = 1 \) to \( \left( \frac{n}{2} \right) \) do

let \( R^j \) be the \( j \)-th subset of \( R \) such that \( |R^j| = i \)

if \( \text{JOINABLE}(R^j) \) then do

let \( \langle R^j_L, R^j_R \rangle \in LR(R^j) \) be a pair of subsets of \( R^j \)

such that \( |\text{COST}(R^j_L) + \text{COST}(R^j_R) + L_j(\text{join}(R^j_L, R^j_R)) \) - \( \text{COMMON}(R^j_L, R^j_R) \) is minimum

\[ \text{COST}(R^j) := \text{COST}(R^j_L) + \text{COST}(R^j_R) + L_j(\text{join}(R^j_L, R^j_R)) \] - \( \text{COMMON}(R^j_L, R^j_R) \)

\[ L(R^j) := R^j_L \]
\[ R(R^j) := R^j_R \]

end

end

\( S := \text{SEQ}(R) \)

end

**Subalgorithm SEQ(R)**

input: a set of relations \( R \), \( L(R) \) and \( R(R) \)
output: the optimal join sequence \( S \) for \( R \)

begin

\( S := \text{null} \)

if \( L(R) = \{ R_{i_1}, R_{i_2} \} \) then \( S := S \oplus \langle R_{i_1}, R_{i_2} \rangle \)

else if \( |L(R)| \geq 3 \) then \( S := S \oplus \text{SEQ}(L(R)) \)

end
if \( R(\mathbf{R}) = \{ R_{i_1}, R_{i_2} \} \) then \( S := S \odot \mathbf{<} R_{i_1}, R_{i_2} \mathbf{>} \)
else if \( |R(\mathbf{R})| \geq 3 \) then \( S := S \odot \mathbf{SEQ}(R(\mathbf{R})) \)
\[ S := S \odot \mathbf{<} \text{join}(L(\mathbf{R})), \text{join}(R(\mathbf{R})) \mathbf{>} \]
\[ S := S \text{ with duplicate join subsequences eliminated} \]
end

4.2. Complexity Analysis

The complexity analysis of algorithm OJS is as follows. The first loop takes \( O(n) \) and the second loop takes \( O(n^2) \). In the worst case, that is, any pair of relations in \( \mathbf{R} \) is joinable, the third loop and forth loop nested in the third take
\[
O \left( \sum_{i=3}^{n} \binom{n}{i} \sum_{j=1}^{i-1} \binom{i}{j} \sum_{k=0}^{j-1} \binom{j}{k} \right) / 2
\]
In spite of the complicated notations, each term can be easily understood. The term \( \binom{n}{i} \) denotes the total number of subsets of \( \mathbf{R} \) with \( i \) relations. For such a subset \( \mathbf{R}^i \), the term \( \binom{i}{j} \) denotes the total number of proper subsets of \( \mathbf{R}^i \) with \( j \) relations, \( 1 \leq j \leq i-1 \), and for such a subset \( \mathbf{R}_L^i \), the term \( \sum_{k=0}^{j-1} \binom{j}{k} \) denotes the total number of subsets of \( \mathbf{R}^i \), one of which is \( \mathbf{R}_R^i \), such that \( \mathbf{<} \mathbf{R}_L^i, \mathbf{R}_R^i \mathbf{>} \in LR(\mathbf{R}^i) \). \( \mathbf{<} \mathbf{R}_L^i, \mathbf{R}_R^i \mathbf{>} \) is the same as \( \mathbf{<} \mathbf{R}_R^i, \mathbf{R}_L^i \mathbf{>} \). Thus, the division by 2. That is, the above complexity is
\[
O \left( \sum_{i=3}^{n} \binom{n}{i} | LR(\mathbf{R}^i) | \right)
\]
The subalgorithm \( \mathbf{SEQ} \) takes a polynomial time. In all, algorithm OJS runs in an exponential time.

4.3. Example

Consider a query \( Q: [R_1.A = R_2.A \land R_2.A = R_3.A \land R_3.B = R_4.B] \) (see Figure 2(a)).
For convenience, $R_i$ is denoted as $i$, $i = 1, 2, 3, 4$ and $\text{join} (R_{i_1}, \ldots, R_{i_j})$ is denoted as $j(i_1, \ldots, i_j)$, $j \geq 2$. Suppose the following information on the sizes of the relations and on the estimated sizes of the intermediate join results is available.

\[ s(1) = 40, \ s(2) = 5, \ s(3) = 50, \ s(4) = 25 \]

\[ s(j(1,2)) = 10, \ s(j(2,3)) = 15, \ s(j(3,1)) = 150, \ s(j(3,4)) = 100 \]

\[ s(j(1,2,3)) = 60, \ s(j(2,3,4)) = 20, \ s(j(1,3,4)) = 500 \]

Without loss of generality, let $w = 1$ and $l_0 = 1$. Then, algorithm $OJS$ obtains the optimal join sequence for $Q$ as follows.

\[
\text{COST}([i]) = 0, \ i = 1, 2, 3, 4
\]

\[
\text{COST}([1,2]) = L_j(1,2) = w \ l_0 \ s(1) \ s(2) = 200
\]

\[
\text{COST}([2,3]) = 250
\]

\[
\text{COST}([3,1]) = 2000
\]

\[
\text{COST}([3,4]) = 1250
\]

\[
\text{COST}([1,2,3]) = \min \{ \text{<COST}([1,2]) + \text{COST}([3]) + L_j(j(1,2,3))>,
\text{<COST}([2,3]) + \text{COST}([1]) + L_j(j(2,3,1))>,
\text{<COST}([3,1]) + \text{COST}([2]) + L_j(j(3,1,2))>,
*\text{<COST}([1,2]) + \text{COST}([2,3]) + L_j(j(1,2),j(2,3))>,
\text{<COST}([2,3]) + \text{COST}([3,1]) + L_j(j(2,3),j(3,1))>,
\text{<COST}([3,1]) + \text{COST}([1,2]) + L_j(j(3,1),j(1,2))> \}
\]

\[
= \min \{ 700, 850, 2750, *600, 4500, 3700 \}
\]

\[
= 600
\]
The minimum is achieved by the pair of subsets $\langle\{1,2\},\{2,3\}\rangle$ of $\{1,2,3\}$, which is marked by *.

Thus,

$L(\{1,2,3\}) = \{1,2\}$

$R(\{1,2,3\}) = \{2,3\}$

\[
\text{COST}(\{2,3,4\}) = \min \left\{ \begin{array}{l}
\ast < \text{COST}(\{2\}) + \text{COST}(\{3,4\}) + L_j(2,3,j(3,4)) >, \\
< \text{COST}(\{2\}) + \text{COST}(\{3,4\}) + L_j(2,3,j(3,4)) >, \\
< \text{COST}(\{2,3\}) + \text{COST}(\{3,4\}) + L_j(2,3,j(3,4)) > \end{array} \right. \\
= \min \left\{ \begin{array}{l}
\ast 625, 1750, 3000 \\
625 \\
\end{array} \right. \\
= 625
\]

$L(\{2,3,4\}) = \{2,3\}$

$R(\{2,3,4\}) = \{4\}$

\[
\text{COST}(\{1,3,4\}) = \min \left\{ \begin{array}{l}
\ast < \text{COST}(\{1\}) + \text{COST}(\{3,4\}) + L_j(1,3,j(3,4)) >, \\
< \text{COST}(\{1\}) + \text{COST}(\{3,4\}) + L_j(1,3,j(3,4)) >, \\
< \text{COST}(\{1,3\}) + \text{COST}(\{3,4\}) + L_j(1,3,j(3,4)) > \end{array} \right. \\
= \min \left\{ \begin{array}{l}
5750, 5250, 18250 \\
5250 \\
\end{array} \right. \\
= 5250
\]

$L(\{1,3,4\}) = \{3,4\}$

$R(\{1,3,4\}) = \{1\}$

\[
\text{COST}(\{1,2,3,4\}) = \min \left\{ \begin{array}{l}
\ast < \text{COST}(\{1,2\}) + \text{COST}(\{3,4\}) + L_j(1,2,j(3,4)) >, \\
< \text{COST}(\{1,2\}) + \text{COST}(\{3,4\}) + L_j(1,2,j(3,4)) >, \\
< \text{COST}(\{1,2,3\}) + \text{COST}(\{4\}) + L_j(1,2,3,4) >, \\
\end{array} \right. \\
\]

-13-
< \text{COST}(\{1\}) + \text{COST}(\{2,3,4\}) + L_j(1,j(2,3,4)) >,
< \text{COST}(\{1,3,4\}) + \text{COST}(\{2\}) + L_j(1,j(1,3,4),2) >,
< \text{COST}(\{1,2,3\}) + \text{COST}(\{3,4\}) + L_j(j(1,2,3),j(3,4)) >,
< \text{COST}(\{1,2\}) + \text{COST}(\{2,3,4\}) + L_j(j(1,2),j(2,3,4)) >,
< \text{COST}(\{1,3\}) + \text{COST}(\{2,3,4\}) + L_j(j(1,3),j(2,3,4)) >,
< \text{COST}(\{1,2\}) + \text{COST}(\{1,3,4\}) + L_j(j(1,2),j(1,3,4)) >,
< \text{COST}(\{1,3,4\}) + \text{COST}(\{2,3\}) + L_j(j(1,3,4),j(2,3)) >,
< \text{COST}(\{1,2,3\}) + \text{COST}(\{2,3,4\}) + L_j(j(1,2,3),j(2,3,4)) - L_j(2,3) >,
< \text{COST}(\{1,2,3\}) + \text{COST}(\{1,3,4\}) + L_j(j(1,2,3),j(1,3,4)) >,
< \text{COST}(\{1,3,4\}) + \text{COST}(\{2,3,4\}) + L_j(j(1,3,4),j(2,3,4)) > |
= \min \{ 2450, 2100, 1425, 7750, 7850, *1025, 5625, 10450, 13000, 2175, 35850, 15875 \}
= 1025

L(\{1,2,3,4\}) = \{1,2\}
R(\{1,2,3,4\}) = \{2,3,4\}

\hline

Thus, the optimal join sequence for Q (see Figure2(b)) is
\quad <1,2> <2,3> <j(2,3),4> <j(1,2),j(2,3,4)> 

5. Heuristic Algorithm

5.1. Algorithm HJS

Algorithm OJS which generates the optimal join sequence for a given query runs in an exponential time. In this section, we develop a simple heuristic algorithm which generates a join sequence in a polynomial time. For a set of relations R referenced in a query, heuristic algorithm HJS generates a join sequence that can be view as a binary
tree in which the leaf nodes are the relations in \( R \), the root node is \( \text{join}(R) \) and the remaining nodes are intermediate join results. The heuristic used in Algorithm HJS is to join smaller relations which generate a smaller join result prior to bigger counterparts.

**Algorithm HJS**

**input**: a set of \( n \) relations \( R \) referenced in a query  
**output**: a join sequence \( S \)

**begin**

\[ S := \text{null} \]

**repeat**

for each joinable pair \( <R_i, R_j> \) in \( R \), compute \( s_p(<R_i, R_j>) \) defined as 
\[ s_p(<R_i, R_j>) = s(R_i) + s(R_j) + s(\text{join}(R_i, R_j)) \]

choose the pair \( <R_i, R_j> \) where \( s_p(<R_i, R_j>) \) is the minimum among all joinable pairs in \( R \)

\[ S := S \oplus <R_i, R_j> \]

\[ R := R \cup \{ \text{join}(R_i, R_j) \} - \{ R_i, R_j \} \]

until \( |R| = 1 \)

**end**

The complexity of algorithm HJS is \( O(n^2) \).

### 5.2. Example

Consider the same query \( Q: [R_1.A = R_2.A \land R_2.A = R_3.A \land R_3.B = R_4.B] \) used in the example for algorithm OJS in section 4.3. The sizes of the relations and the intermediate join results are

\[ s(1) = 40, \ s(2) = 5, \ s(3) = 50, \ s(4) = 25 \]

\[ s(j(1,2)) = 10, \ s(j(2,3)) = 15, \ s(j(3,1)) = 150, \ s(j(3,4)) = 100 \]

\[ s(j(1,2,3)) = 60, \ s(j(2,3,4)) = 20, \ s(j(1,3,4)) = 500 \]

With \( R = \{1, 2, 3, 4\} \), algorithm HJS obtains a join sequence for \( Q \) as follows.
\( S = null \)

In the first run of the loop,

\[ s_p(\langle 1,2 \rangle) = s(1) + s(2) + s(j(1,2)) = 55 \]
\[ s_p(\langle 2,3 \rangle) = s(2) + s(3) + s(j(2,3)) = 70 \]
\[ s_p(\langle 3,4 \rangle) = s(3) + s(4) + s(j(3,4)) = 175 \]
\[ s_p(\langle 1,3 \rangle) = s(1) + s(3) + s(j(1,3)) = 240 \]

The pair \( \langle 1,2 \rangle \) is chosen.

\( S = \langle 1,2 \rangle \)

\( R = \{ j(1,2), 3, 4 \} \)

In the next run,

\[ s_p(\langle j(1,2),3 \rangle) = s(j(1,2)) + s(3) + s(j(1,2,3)) = 120 \]
\[ s_p(\langle 3,4 \rangle) = s(3) + s(4) + s(j(3,4)) = 175 \]

The pair \( \langle j(1,2),3 \rangle \) is chosen.

\( S = \langle 1,2 \rangle < j(1,2),3 \rangle \)

\( R = \{ j(1,2,3), 4 \} \)

Finally, the pair \( \langle j(1,2,3), 4 \rangle \) is chosen and the algorithm terminates with \( S = \langle 1,2 \rangle < j(1,2),3 \rangle < j(1,2,3),4 \rangle \).

That is, the generated join sequence is \( \langle 1,2 \rangle < j(1,2),3 \rangle < j(1,2,3),4 \rangle \) (see Figure 2(c)).
6. Concluding Remarks

In this paper, we were concerned with the cost-effectiveness of a semijoin in distributed query processing considering the local processing cost as well as the data transmission cost. The cost of a semijoin was a mere sum of the costs of the local processing component and of the data transmission component in executing the semijoin. However, the benefit computation was more complicated for which we introduced the join sequence. We developed algorithm $OJS$ which generates the optimal join sequence for a query. Since algorithm $OJS$ runs in an exponential time, we also developed a simple heuristic algorithm $HJS$ which generates a join sequence for a query in a polynomial time.

In the reducing phase of processing a query, only semijoins are used to reduce relations while joins are delayed until the joining phase. The preference of semijoins to joins for relation reducing is based on heuristics. Many researchers agreed on that the optimal query processing strategy would use both joins and semijoins in the reducing phase [Bernstein et al 81][Gouda and Dayal 81][Kershberg et al 82]. For the reducing phase using both joins and semijoins, the join sequence can still be used, with a simple extension, to measure the effectiveness of joins and/or semijoins in terms of the local processing cost [Kang and Roussopoulos 87].

References

[Apers et al 83]

[Bernstein and Chiu 81]
[Bernstein et al. 81]

[Black and Luk 82]

[Chang 82]
Chang, J., *A Heuristic Approach to Distributed Query Processing*, Proc. 8th Int. Conf. on Very Large Data Bases, 1982.

[Cheung 82]

[Gouda and Dayal 81]

[Hevner 79]

[Hevner and Yao 79]

[Kang and Roussopoulos 87]
Kang, H. and Roussopoulos, N., *Combining Joins and Semijoins in Distributed Query Processing*, Submitted for publication.

[Kerschberg et al. 82]

[Lu and Carey 85]

[Mackert and Lohman 86]

-18-
[Selinger et al 79]

[Wong 77]

[Wong and Youssefi 76]

[Yao 79]

[Yu et al 82]
Figure 1  Join Sequences
(a) query Q

(b) optimal join sequence for Q generated by OJS

(c) join sequence for Q generated by HJS

Figure 2