Combining Joins and Semijoins in Distributed Query Processing
by
Hyunchul Kang
Nick Roussopoulos
Combining Joins and Semijoins in Distributed Query Processing
by
Hyunchul Kang
Nick Roussopoulos
February 1987

Combining Joins and Semijoins 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

A new approach to distributed query processing is proposed. In the conventional approach, a query is processed by the reducing phase and the joining phase. In the reducing phase, the relations referenced in the query are reduced using semijoins, and all the relations are joined at the result site in the joining phase. In the proposed approach, an arbitrary interleaving of joins and semijoins is allowed toward the generation of the query result at the result site. The proposed approach considers the local processing cost as well as the data transmission cost while the conventional approach mostly considers the data transmission cost only. The effectiveness of the query processing strategies generated according to the proposed approach is discussed for both the restricted class of join queries and the general join queries.
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 assume that the local processing cost is negligible compared to the data transmission cost and process a query by the following two phases [Wong 77][Bernstein et al 81][Black and Luk 82][Hevner and Yao 79][Chang 82][Cheung 82][Apers et al 83][Yu et al 82]:

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 query optimization is achieved in the reducing phase where the semijoin program reduces the relations and thus, reduces the data transmission cost. This conventional approach to distributed query processing is limited in two aspects:

(1) In the reducing phase where the query optimization is achieved, only semijoins are used. The preference of semijoins to joins is based on heuristics. Many researchers agreed on that the optimal processing strategy would use both joins and semijoins for query optimization [Bernstein et al 81][Gouda and Dayal 81][Kerschberg et al 82].

(2) 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 even in a geographically distributed environment [Lu and Carey 85][Mackert and Lohman 86]. Since semijoins are reducing the data transmission cost at the expense of the local processing cost, the effectiveness of query processing strategies based on semijoins can be severely damaged if the local processing cost is significant.

In this paper, we propose a new approach to distributed query processing where both joins and semijoins are used for query optimization. That is, we allow an arbitrary interleaving of joins and semijoins toward the generation of the query result at the result site. We also do not assume that the local processing cost is negligible compared to the data transmission cost. Thus, the query optimization is done against the local processing cost as well as the data transmission cost. A query is processed by the following two phases:
(1) \textit{intermediate phase}: executes an \textit{intermediate strategy} (I-strategy) which is a sequence of joins and/or semijoins. The relations are reduced by semijoins and/or joined to generate the intermediate join results.

(2) \textit{final phase}: executes a \textit{final strategy} (F-strategy) which is a sequence of final join strategies. All the relations remaining out of the intermediate phase are joined to answer the query. All the relations are not necessarily sent to the result site. Instead, some can be sent to a site, other than the result site, where they are joined to generate the intermediate join result leading to the query result. This phase is completed when the result site has the query result.

The rest of this paper is organized as follows. Section 2 and 3 are devoted to the generation of I-strategy and of F-strategy for a given query, respectively. Section 4 discusses the effectiveness of the resulting I-strategy and F-strategy for both the restricted and general join queries. Concluding remarks are in section 5.

2. Generation of I-strategy

Given a query, a probable strategy to answer the query is to send all the relations referenced in the query to the result site and join them. We call this strategy as the \textit{feasible strategy}.\textsuperscript{1} The feasible strategy could be improved by executing some semijoins and/or joins among the relations before they are sent to the result site. The semijoins and/or joins executed modify the query. The feasible strategy for the modified query, that is, the strategy which sends all the relations remaining after the semijoins and/or joins are executed to the result site and joins them, could be also improved similarly, and so on.

Example

\textsuperscript{1} This strategy is called as the initial feasible strategy in the literature.
Given query $Q_0$ joining $R_1, R_2, R_3$ and $R_4$, which are at different sites other than the
result site, the feasible strategy is to

send $R_1, R_2, R_3$ and $R_4$ to the result site and join them.

This strategy is improved by executing $\text{semijoin}(R_1 \rightarrow R_2)^2$ before $R_1, R_2, R_3$ and $R_4$ are
sent to the result site if the cost of executing the semijoin followed by sending
$R_1, R_2', R_3$ and $R_4$ to the result site and joining them is less than that of the feasible
strategy. After this semijoin is executed, $Q_0$ is modified to $Q_1$ joining $R_1, R_2', R_3$ and
$R_4$. The feasible strategy for $Q_1$ is improved by executing $\text{semijoin}(R_2' \rightarrow R_3)$ followed by
$\text{join}(R_3' \rightarrow R_2')^3$ before $R_1, R_2', R_3$ and $R_4$ are sent to the result site for the similar rea-
son. After this semijoin and join sequence is executed, $Q_1$ is modified to $Q_2$ joining $R_1,$
$\text{join}(R_2', R_3')^4$ and $R_4$. The feasible strategy for $Q_2$ is improved by executing
$\text{semijoin}(\text{join}(R_2', R_3') \rightarrow R_4)$ before $R_1,$ $\text{join}(R_2', R_3')$ and $R_4$ are sent to the result
site for the similar reason, and so on. $\square$

We develop an algorithm, Algorithm I, which generates I-strategy which is a sequence of
joins and/or semijoins using the above tactic for a given query. It starts with null I-strategy. It selects a pair of joinable relations $< R_i, R_j >$ and chooses a $\text{semijoin and/or join sequence}$ between $R_i$ and $R_j$ $^5$ such that the chosen sequence improves the feasible
strategy for the query most. The query is modified and Algorithm I chooses a semijoin
and/or join sequence between another pair of joinable relations for the modified query

\footnote{The notation $\text{semijoin}(R_i \rightarrow R_j)$ denotes that the joining attribute values of $R_i$ are sent to the site of $R_j$ to reduce $R_j$ into $R_j'$.}

\footnote{The notation $\text{join}(R_i \rightarrow R_j)$ denotes that $R_i$ is sent to the site of $R_j$ for the join.}

\footnote{$\text{join}(R_i, R_j)$ is the join result of $R_i$ and $R_j$.}

\footnote{Some examples of the semijoin and/or join sequence between $R_i$ and $R_j$ are $\text{semijoin}(R_i \rightarrow R_j)$ or $\text{semijoin}(R_i \rightarrow R_j)$ followed by $\text{semijoin}(R_i', R_j')$ followed by $\text{join}(R_i', R_j')$, and so on. The complete list of these sequences will be developed in subsection 2.1.}
most, and so on. Every time a semijoin and/or join sequence is chosen, I-strategy is expanded by appending the chosen sequence at the end of the current I-strategy. This process is repeated until either no more improvement can be achieved or the query result is generated.

The rest of this section is organized as follows. To present Algorithm I, we have two subsections. In subsection 2.1, we develop the semijoin and/or join sequences between two joinable relations. These sequences will be used to expand I-strategy by Algorithm I. Since Algorithm I repeatedly tries to improve the feasible strategy for the query, it is necessary to measure the cost to execute the feasible strategy. In subsection 2.2, we introduce join tree to measure the local processing cost of the feasible strategy to join the relations at the result site. Subsection 2.3 presents Algorithm I. Finally, subsection 2.4 describes how a pair of joinable relations to be joined and/or semijoined is selected.

2.1. Semijoin and/or Join Sequence

Given a pair of joinable relations $R_1$ and $R_2$ at different sites, there are at most 10 different semijoin and/or join sequences that can be considered between $R_1$ and $R_2$.

$\text{sequence}_0$: null.

This sequence neither reduces $R_1$ or $R_2$ by semijoins nor joins the two.

$\text{sequence}_1$: semijoin($R_1 \rightarrow R_2$).

$\text{sequence}_2$: join($R_1 \leftarrow \rightarrow R_2$).

Let $S(R)$ be the size of relation $R$. Then, the notation join($R_i \leftarrow \rightarrow R_j$) denotes that

1. either $R_i$ or $R_j$, which is less in size is sent to the site of the other and they are joined.
(2)  \[ S(\text{join}(R_i, R_j)) \leq \max\{ S(R_i), S(R_j) \} \]

When a join is to be processed by sending one relation to the site of the other, there are two conditions to be satisfied which are specified in (1) and (2) above. The first condition is based on the assumption that the data transmission cost is a linear function of the data volume to be sent. That is, the cost to send \( x \) amount of data is \( c_0 + c_1x \) where \( c_0 \) and \( c_1 \) are constants. This cost measurement is widely used in the literature. As for the second condition, if it is not satisfied, such join is not feasible because it would incur less data transmission cost to send both relations separately to a destination site where the join result is needed and join them there to join at the site of one relation and send the join result to the destination site. When \( S(R_i) \leq S(R_j) \) is known, the notation join\((R_i \rightarrow R_j)\) will be used.

Once either \( R_1 \) or \( R_2 \) is sent to the site of the other, the joining of \( R_1 \) and \( R_2 \) is a local join. When a local join is to be processed, some join filtering can be preceded [Yao 79].

The join filtering eliminates tuples which are not joinable from one or both relations. If we consider the join filtering by local semijoins, 5 different join filtering sequences can be preceded before \( R_1 \) and \( R_2 \) are joined: (1) no filtering (2) \( f(R_1 \rightarrow R_2) \) where \( f(R_i \rightarrow R_j) \) denotes the filtering of \( R_j \) using joining attribute values of \( R_i \). \( R_j \) is denoted as \( R_j' \) after filtering. (3) \( f(R_1 \rightarrow R_2) \) followed by \( f(R_2' \rightarrow R_1) \) (4) \( f(R_2 \rightarrow R_1) \) and (5) \( f(R_2 \rightarrow R_1) \) followed by \( f(R_1' \rightarrow R_2) \). When a local join is to be processed, we assume that some join filtering sequence is preceded in order to minimize the local processing cost.

sequence 3: \{ semijoin\((R_1 \rightarrow R_2)\) followed by join\((R_1 \leftarrow \rightarrow R_2'\)) \}.

The notation \{ semijoin\((R_i \rightarrow R_j)\) followed by join\((R_i \leftarrow \rightarrow R_j'\)) \} denotes

\[
\begin{align*}
\text{join}(R_i \rightarrow R_j) & \quad \text{if } S(R_i) \leq S(R_j') \\
\text{semijoin}(R_i \rightarrow R_j) \text{ followed by } \text{join}(R_j' \rightarrow R_i) & \quad \text{otherwise}
\end{align*}
\]
If \( S(R_1) \leq S(R'_2) \), \( \text{join}(R_1 \leftarrow R'_1) \) is \( \text{join}(R_1 \leftarrow R'_2) \). In sequence \( \text{semijoin}(R_1 \leftarrow R_2) \) followed by \( \text{join}(R_1 \leftarrow R'_2) \), the joining attribute values of \( R_1 \) are sent redundantly in generating \( \text{join}(R_1, R_2) \). Thus, \( \text{semijoin}(R_1 \leftarrow R_2) \) followed by \( \text{join}(R_1 \leftarrow R'_2) \) is replaced by \( \text{join}(R_1 \leftarrow R_2) \).

\[ \text{sequence}_4: \] \( \text{semijoin}(R_1 \leftarrow R_2) \) followed by \( \text{semijoin}(R'_2 \leftarrow R_1) \).

\[ \text{sequence}_5: \] \( \text{semijoin}(R_1 \leftarrow R_2) \) followed by

\[ \{ \text{semijoin}(R'_2 \leftarrow R_1) \text{ followed by } \text{join}(R_1' \leftarrow R'_2) \} \].

\[ \text{sequence}_6: \] \( \text{semijoin}(R_2 \leftarrow R_1) \).

\[ \text{sequence}_7: \] \( \{ \text{semijoin}(R_2 \leftarrow R_1) \text{ followed by } \text{join}(R_1' \leftarrow R_2) \} \).

\[ \text{sequence}_8: \] \( \text{semijoin}(R_2 \leftarrow R_1) \) followed by \( \text{semijoin}(R_1' \leftarrow R_2) \).

\[ \text{sequence}_9: \] \( \text{semijoin}(R_2 \leftarrow R_1) \) followed by

\[ \{ \text{semijoin}(R_1' \leftarrow R_2) \text{ followed by } \text{join}(R_1' \leftarrow R'_2) \} \].

2.2. Join Tree

Given a query, the feasible strategy joins all the relations referenced in the query at the result site. We call the sequence in which the relations are joined by the feasible strategy at the result site as the join sequence. For a query, there may be more than one different join sequence at the result site.

Example

Given a query joining \( R_1, R_2, R_3, \) and \( R_4 \), if any pair of relations is joinable, the relations can be joined in the sequence, join \( R_1 \) and \( R_2 \), join \( \text{join}(R_1, R_2) \) and \( R_3 \), join \( \text{join}(R_1, R_2, R_3) \) and \( R_4 \) (see Figure 1a) or in the sequence, join \( R_2 \) and \( R_3 \), join \( R_1 \) and \( \text{join}(R_2, R_3) \), join \( \text{join}(R_1, R_2, R_3) \) and \( R_4 \) (see Figure 1b), and so on. □
The local processing cost of joining relations according to a join sequence is different from sequence to sequence. Since the local processing cost is not negligible, the feasible strategy in the literature needs to be extended by specifying a particular join sequence at the result site. Given a query, the optimal join sequence at the result site can be generated by a dynamic programming algorithm [Kang and Roussopoulos 87a]. Since this algorithm runs in an exponential time, we develop a heuristic algorithm, Algorithm JT, which generates a join sequence in a polynomial time. The join sequence generated by Algorithm JT can be represented as a binary tree where the leaf nodes are the relations referenced in the query, the root node is the query result and the remaining nodes are intermediate join results. We call this join sequence as the join tree. The heuristic used in Algorithm JT is to join smaller relations which generate a smaller join result prior to bigger counterparts. This heuristic is based on the assumption that the local processing cost of joining smaller relations is less than that of joining bigger ones.\(^6\) Algorithm JT is presented in Appendix.

2.3. Algorithm I

Given a query, Algorithm I starts with null I-strategy and a join tree constructed by Algorithm JT. I-strategy is expanded repeatedly by appending some semijoin and/or join sequence between a pair of joinable relations. The join tree is used to measure the local processing cost of the feasible strategy for the query to join the relations at the result site.

I-strategy Generation Algorithm I

input: a query

---

\(^6\) The detailed discussion on the local processing cost in distributed query processing is in [Kang and Roussopoulos 87a].
output: \textit{I-strategy}

(1) \textit{I-strategy} := \text{null}.

(2) Construct the join tree $T_J$ of the query by \textit{Algorithm JT}.

(3) Select a pair of joinable relations <\textit{R}_1, \textit{R}_2>.\footnote{Subsection 2.4 will describe how a pair of joinable relations is selected.}

(4) Consider the 10 semijoin and/or join sequences between \textit{R}_1 and \textit{R}_2 and choose \textit{sequence}_i such that

$$C(\text{sequence}_i) = \min_{0 \leq i \leq 9} \left[ C(\text{sequence}_i) \right]$$

where

$$C(\text{sequence}_i) = C_1(\text{sequence}_i) + C_2(\text{sequence}_i) + C_3(\text{sequence}_i, T_J), \ i = 0, \ldots, 9.$$  

\textit{C}_1(\text{sequence}_i):  

the data transmission and the local processing cost to execute the semijoins and/or join in \textit{sequence}_i.

\textit{C}_2(\text{sequence}_i):  

the data transmission cost of sending the results of \textit{sequence}_i, executed between \textit{R}_1 and \textit{R}_2 (either \textit{join}(\textit{R}_1, \textit{R}_2) or possibly reduced \textit{R}_1 and \textit{R}_2) and all other relations referenced in the query to the result site.

\textit{C}_3(\text{sequence}_i, T_J):  

the local processing cost of joining all the relations at the result site according to join tree $T_J$, which is constructed by modifying $T_J$ to reflect the effects of \textit{sequence}_i, executed between \textit{R}_1 and \textit{R}_2.
(5) \textit{I-strategy} := \textit{I-strategy} followed by sequence$_j$.

(6) Modify the query by reflecting the effects of sequence$_j$ executed between $R_1$ and $R_2$.

(7) Replace $T_j$ with $T_j'$. 

(8) Repeat step (3) through (7) until either no further pair of joinable relations can be selected or the query result is generated.

In step (4), each semijoin and/or join sequence considered would modify the query if chosen. The join tree need be also modified by reflecting the effects of the semijoin and/or join sequence considered to measure the local processing cost of the feasible strategy for the modified query.

\textbf{Example}

Given a query joining $R_1, R_2, R_3$ and $R_4$, suppose \textit{Algorithm JT} constructed the join tree, join $R_2$ and $R_3$, join $R_1$ and $\text{join}(R_2, R_3)$, join $\text{join}(R_1, R_2, R_3)$ and $R_4$ (see Figure 1b).

When \textit{I-strategy} is appended by semijoin($R_1 \rightarrow R_2$), the join tree is modified to join $R_2'$ and $R_3$, join $R_1$ and $\text{join}(R_2', R_3)$, join $\text{join}(R_1, R_2', R_3)$ and $R_4$ (see Figure 2a).

When \textit{I-strategy} is appended by semijoin($R_2' \rightarrow R_3$) followed by join($R_3' \rightarrow R_2'$), the join tree is modified to join $R_1$ and $\text{join}(R_2', R_3')$, join $\text{join}(R_1, R_2', R_3')$ and $R_4$ (see Figure 2b), and so on. □

We have an algorithm, \textit{Algorithm MJT}, which modifies the join tree to reflect the effects of a semijoin and/or join sequence using the same heuristic used in \textit{Algorithm JT}. \textit{Algorithm MJT} is presented in Appendix.
2.4. Selection of Joinable Pair of Relations

In generating \textit{I-strategy}, the order in which pairs of joinable relations are selected has a great influence on the effectiveness of the resulting \textit{I-strategy}. \textit{I-strategy} is a sequence of joins and/or semijoins. It is desirable to schedule joins and/or semijoins in \textit{I-strategy} in such a way that the smaller relations (that is, the relations with the smaller relation size and/or the smaller joining attribute size) are joined and/or semijoinied prior to the bigger relations.

We order relations in the ascending order of the relation size, and for each set of relations with common joining attributes, we order relations in the set in the ascending order of the joining attribute size. With these orderings, for each relation \( R \), we compute \( \text{order}(R) \) where \( \text{order} \) is the function of \( R \) defined as follows.

For relation \( R_i \), let \( m_i \) be the number of different orderings where \( R_i \) participates. \( m_i \) is at least 2, one for the relation size ordering, the other for the joining attribute size ordering. Let \( \text{position}(i,j) \) be the position of \( R_i \) in the \( j \)-th ordering, \( 1 \leq j \leq m_i \). That is, if \( R_i \) is at the \( k \)-th position in the \( j \)-th ordering, then \( \text{position}(i,j) = k \). Then, the function \( \text{order} \) is defined as follows:

\[
\text{order}(R_i) = \frac{\sum_{j=1}^{m_i} \text{position}(i,j)}{m_i}
\]

We have an algorithm, \textit{Algorithm SP}, which selects a pair of joinable relations to be joined and/or semijoinied by \textit{I-strategy}. It first orders the relations in the ascending order of their \( \text{order} \) values, and picks two relations which are joinable with each other occurring as early as possible in the ordering. \textit{Algorithm SP} is presented in Appendix.
3. Generation of F-strategy

Given a query, I-strategy is generated first by Algorithm I. The execution of I-strategy would modify the query. For this modified query, F-strategy is generated next which is a sequence of final join strategies toward the generation of the query result at the result site. A probable F-strategy is to send all the relations remaining in the query to the result site and join them. That is, the feasible strategy for the modified query. This strategy could not be improved by executing some semijoin and/or join sequence between a pair of joinable relations. Otherwise, I-strategy would have been further expanded. However, this feasible strategy could be improved by joining more than two relations at a site other than the result site and sending the join result to the result site.

Example

Given a query joining $R_1, R_2, R_3$ and $R_4$, which are at different sites other than the result site, suppose Algorithm I generated I-strategy:

semijoin($R_1 \rightarrow R_2$), semijoin($R_2' \rightarrow R_3$), join($R_4' \rightarrow R_4$), semijoin(join($R_2', R_3') \rightarrow R_4$).

Then, the modified query after I-strategy is executed is to join $R_1$, join($R_2', R_3'$) and $R_4'$. The feasible strategy for this modified query is to

send $R_1$, join($R_2', R_3'$) and $R_4'$ to the result site and join them.

This strategy is improved by sending $R_1$ and $R_4'$ to the site of join($R_2', R_3'$) where the three are joined to generate the query result join($R_1, R_2, R_3, R_4$) and sending join($R_1, R_2, R_3, R_4$) to the result site if join($R_1, R_2, R_3, R_4$) is less than join($R_2', R_3'$) in size. Since the local processing cost to join $R_1$, join($R_2', R_3'$) and $R_4'$ is the same regardless of the site where the join is computed, the improvement is in the data transmission cost between sending join($R_2', R_3'$) and join($R_1, R_2, R_3, R_4$). □
The joins at a site other than the result site modifies the query further. The feasible strategy for this modified query could be also improved similarly. We develop an algorithm, Algorithm $F$, which generates $F$-strategy using the above tactic for the query modified by $I$-strategy. It starts with null $F$-strategy. It constructs a join tree for the modified query by Algorithm $JT$. Starting from the smaller to larger subtree of the join tree, it considers a subtree with relations $R_1, ..., R_m$ as leaf nodes. If $S(\text{join}(R_1, ..., R_m)) \leq \max\{S(R_1), ..., S(R_m)\}$, then $F$-strategy is expanded by appending the final join strategy which sends $R_1, ..., R_{m-1}$ to the site of $R_m$ and joins them where $S(R_m) = \max\{S(R_1), ..., S(R_m)\}$, at the end of the current $F$-strategy. The join tree is modified to reflect the effects of the appended final join strategy, and Algorithm $F$ considers another subtree of the modified join tree to expand $F$-strategy further. This process is repeated until either no more improvement can be achieved or the query result is generated. In the former case, $F$-strategy is appended by the feasible strategy for the remaining query, and in the latter, $F$-strategy is appended by the strategy merely sending the query result to the result site.

**F-strategy Generation Algorithm F**

input: the query modified by $I$-strategy

output: $F$-strategy

1. $F$-strategy := null.

2. If the query result $R_s$ has been generated by $I$-strategy, then

   $F$-strategy := $F$-strategy followed by $F$-strategy($R_s$)

   where $F$-strategy($R_s$) sends $R_s$ to the result site.

   Otherwise, do the following steps.

---

*Or it can use the join tree left after $I$-strategy is generated.*
(3) Construct the join tree $T_J$ of the modified query by Algorithm JT.

(4) For each node $R_r$ of $T_J$ visited in the postorder traversal\(^8\), do step (5).

(5) Let $T$ be the subtree of $T_J$ with $R_r$ as the root node. Let $R_1, \ldots, R_m$ be the leaf nodes of $T$.

If $S(\text{join}(R_1, \ldots, R_m)) \leq \max[ S(R_1), \ldots, S(R_m) ]$, then

$$F\text{-strategy} := F\text{-strategy followed by } F\text{-strategy}(T, R_m)$$

and modify $T_J$ by replacing $T$ with $R_r$.

where $F\text{-strategy}(T, R_m)$ sends $R_1, \ldots, R_{m-1}$ to the site of $R_m$ and joins them according to $T$ where $S(R_m) = \max[ S(R_1), \ldots, S(R_m) ]$.

(6) If $T_J$ consists of one node $R_r$, then

$$F\text{-strategy} := F\text{-strategy followed by } F\text{-strategy}(R_r)$$

(7) If $T_J$ is a tree with $R_1, \ldots, R_l$ as leaf nodes, then

$$F\text{-strategy} := F\text{-strategy followed by } F\text{-strategy}(T_J, R_1, \ldots, R_l)$$

where $F\text{-strategy}(T_J, R_1, \ldots, R_l)$ sends $R_1, \ldots, R_l$ to the result site and joins them according to $T_J$.

4. Evaluation of Algorithm I and F

In this section, the effectiveness of the query processing strategy ($I\text{-strategy followed by } F\text{-strategy}$) generated by Algorithm I and $F$ is discussed. We consider first some important class of restricted join queries, single join query and simple query, and then the general join query. The proofs of the lemmas below are in [Kang and Roussopoulos 87b].

---

\(^8\) The postorder traversal of a binary tree is defined recursively as follows [Horowitz and Sahni 76]:

(1) the left subtree is traversed in the postorder.
(2) the right subtree is traversed in the postorder.
(3) the root node is visited.
4.1. Single Join Query

A single join query is a query joining two relations at different sites. That is, the query result is the single join result.

Lemma 1

For a single join query, Algorithm I and F generate the optimal strategy. □

4.2. Simple Query

A simple query is a query joining relations each of which consists of one attribute which is common to all relations [Hevner 79]. The optimal strategy for a simple query is given by Hevner and Yao [1979] when ignoring the local processing cost. Suppose n relations are joined such that \( |R_1| \leq ... \leq |R_n| \). Then, the optimal strategy which minimizes the total data transmission cost is

\[
\text{join}(R_1 \rightarrow ... \rightarrow R_n)
\]

which denotes join\((R_1 \rightarrow R_2)\) followed by join\((\text{join}(R_1,R_2) \rightarrow R_3)\) followed by join\((\text{join}(R_1,R_2,R_3) \rightarrow R_4)\) and so on. This strategy can be shown to be also the optimal strategy when considering both the local processing cost and the data transmission cost.

We have the following two lemmas.

Lemma 2

Assume that the local processing cost for joining two single attribute relations \( R_i \) and \( R_j \) is proportional to \( |R_i| |R_j| \). Then for a simple query joining \( n \) relations such that \( |R_1| \leq ... \leq |R_n| \), the optimal strategy when considering both the local processing cost and the data transmission cost is

\[
\text{join}(R_1 \rightarrow ... \rightarrow R_n). \quad \Box
\]

Lemma 3

For a simple query, Algorithm I and F generate the optimal strategy. □
4.3. General Join Query

Given a query, Algorithm I and F generate the processing strategy, strategy, such that

\[ \text{strategy} = \text{I-strategy} \text{ followed by F-strategy} \]

Algorithm I expands I-strategy, which is initially null, by appending substrategies one after another. Each substrategy is a semijoin and/or join sequence between a pair of joinable relations. Similarly, Algorithm F expands F-strategy, which is initially null, by appending substrategies one after another. Each substrategy is a final join strategy toward the generation of the query result at the result site. Thus for some \( n \geq 1 \),

\[ \text{strategy} = \text{substrategy}_1 \text{ followed by } \ldots \text{ followed by } \text{substrategy}_n \]

It has been proved that the problem of generating the optimal processing strategy for a general join query is NP-hard [Hevner 79][Yu et al 82]. Thus, the query optimization algorithms for general join queries in the literature depend on heuristics. The query processing strategy generated by a heuristic algorithm is, in general, suboptimal.

Since Algorithm I and F are such heuristic algorithms, although they seek the locally optimal expansion of I-strategy and F-strategy, the resulting query processing strategy (I-strategy followed by F-strategy) may be globally suboptimal for a general join query. However globally, Algorithm I and F maintain the property that every time I-strategy or F-strategy is expanded by appending a substrategy, the expansion gives improvement over the current feasible strategy for the query. This property is elaborated below. Let

\[ \text{strategy}(0) = \text{null}. \]

\[ \text{strategy}(i) = \text{substrategy}_1 \text{ followed by } \ldots \text{ followed by } \text{substrategy}_i, \quad i \leq n. \]

Given general join query \( Q_0 \), let query \( Q_i \) be such that

\[ Q_i = \text{strategy}(i)(Q_0), \quad 0 \leq i \leq n. \]

That is, \( Q_i \) is the query modified after \( \text{strategy}(i) \) is executed for \( Q_0 \). Let \( FS(i) \) be the
feasible strategy for $Q_i$. We have the following lemma.

**Lemma 4**

Let $cost(\text{STRATEGY}(i))$ ($cost(\text{FS}(i))$) be the local processing and data transmission cost to execute $\text{STRATEGY}(i)$ ($\text{FS}(i)$), $0 \leq i \leq n$. Then for a general join query,

$$cost(\text{STRATEGY}(i)) + cost(\text{FS}(i)) \geq cost(\text{STRATEGY}(i+1)) + cost(\text{FS}(i+1)),$$  

$0 \leq i \leq n$. □

5. Concluding Remarks

The effectiveness of the processing strategies generated by *Algorithm I* and $F$ for general join queries need be further evaluated. Some simulation is planned for this purpose.

The query processing strategy generated by a heuristic algorithm is, in general, suboptimal. Assuming that the local processing cost is negligible compared to the data transmission cost, the properties that an optimal semijoin program should possess have been studied [Chen and Li 83] and the algorithms for improving semijoin programs have been proposed [Chen and Li 84][Luk and Luk 83][Bernstein et al 81][Yu and Chang 83]. Most of these improving techniques for semijoin programs are applicable to query processing strategies generated by *Algorithm I* and $F$. However, the fact that the local processing cost is not ignored and that joins and semijoins are combined in the query processing strategy needs further exploration.

It is important to estimate the size of the projection of joining attributes with duplicates eliminated which is to be used in a semijoin and to estimate the size of a semijoin result and a join result. The effectiveness of the query processing strategies depends on the correctness of these estimations. These estimation techniques are proposed in the literature [Bernstein et al 81][Luk and Black 81][Selinger et al 79][Yao 79]. However, further works need be done to estimate the effects of an arbitrary interleaving of joins and semijoins.
Appendix

Join Tree Construction Algorithm JT

input: a query joining $R_1,\ldots,R_n$

output: join tree $T_J$

(1) Create nodes $R_1,\ldots,R_n$.

(2) $R := \{ R_1,\ldots,R_n \}$.

(3) Repeat the following steps until $|R| = 1$.

(4) For each joinable pair $<R_i,R_j>$ in $R$, compute $S_p(<R_i,R_j>)$ defined as

$$S_p(<R_i,R_j>) = S(R_i) + S(R_j) + S(\text{join}(R_i,R_j)).$$

(5) Choose the pair $<R_i,R_j>$ where $S_p(<R_i,R_j>)$ is the minimum among all joinable pairs in $R$.

(6) Create a node $R_\ast$ such that $R_\ast$ is the parent of $R_i$ and $R_j$.

(7) $R := R \cup \{ R_\ast \} - \{ R_i,R_j \}$.

Join Tree Modification Algorithm MJT

input: join tree $T_J$, semijoin and/or join sequence $sequence_i$ executed between $R_i$ and $R_j$, $i = 0,\ldots,9$.

output: modified join tree $T_J'$

(1) If $sequence_i$ does not join $R_i$ and $R_j$, then $T_J' := T_J$ with $R_i$ and $R_j$ replaced by $R_i'$ and $R_j'$, respectively if they are reduced by semijoins.

(2) If ($sequence_i$ joins $R_i$ and $R_j$) \& (R_i and R_j are siblings in T_J), then

$$T_J' := T_J \text{ with } R_i \text{ and } R_j \text{ removed}.$$ 

(3) If ($sequence_i$ joins $R_i$ and $R_j$) \& (R_i and R_j are not siblings in T_J), then

$$T_J' := T_J \text{ with adjustments done in the following steps.}$$
(1) Remove $R_i$ and $R_j$.

(2) Let $R_i^*$ and $R_j^*$ be the siblings of $R_i$ and $R_j$, respectively in $T_J$. Without loss of generality, suppose $S_p(<R_{i^*},join(R_i,R_j)>)$ $\leq$ $S_p(<R_{j^*},join(R_i,R_j)>).$ Create a node $R_k$ where $R_k = join(R_i,R_j)$ such that $R_k$ becomes the sibling of $R_i^*$.

(3) Remove the parent of $R_j^*$ such that the grandparent of $R_j^*$, if there is one, becomes the parent of $R_j^*$.

**Pair of Joinable Relations Selection Algorithm SP**

input: $n$ relations to be joined

output: a pair of joinable relations

(1) For each relation $R$, compute $order(R)$.

(2) Order relations in the ascending order of their $order$ values. Rename relations such that $order(R_1) \leq ... \leq order(R_n)$.

(3) $p := <R_1,R_2>.$

(4) If the two relations in $p$ are joinable and $p$ was not selected before, then $p$ is selected.

(5) If $p = <R_i,R_j>$ is not selected, then $p := <R_{i+1},R_j>.$

(6) If $i + 1 = j = n$, then no further pair can be selected.

(7) If $i + 1 = j < n$, then $p := <R_1,R_{j+1}>.$

(8) Go to step (4).

**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.

[Chen and Li 83]

[Chen and Li 84]

[Cheung 82]

[Gouda and Dayal 81]

[Hevner 79]

[Hevner and Yao 79]

[Horowitz and Sahni 76]

[Kang and Roussopoulos 87a]

[Kang and Roussopoulos 87b]

[Kerschberg et al 82]
[Lu and Carey 85]

[Luk and Black 81]

[Luk and Luk 83]

[Mackert and Lohman 86]
Mackert, L. and Lohman, G., *R* *Optimizer Validation and Performance Evaluation for Distributed Queries*, Proc. 12-th Int. Conf. on Very Large Data Bases, 1986.

[Selinger et al 79]

[Wong 77]

[Wong and Youssefi 76]

[Yao 79]

[Yu and Chang 83]

[Yu et al 82]
Figure 1. Join Sequences

Figure 2. Join Tree Modification