

A Distributed Query Processing Strategy Using Placement Dependency*

Chengwen Liu, Hao Chen and Warren Krueger

School of Computer Science, Telecommunications and Information Systems

DePaul University, Chicago, Illinois 60604

Abstract

We present an algorithm to make use of placement dependency information to process distributed queries. Our algorithm first partitions the referenced relations of a given query into a number of non-exclusive subsets such that the fragmented relations within a subset have placement dependency and the join operation(s) associated with the relations in the subset can be locally processed without data transfer. Each subset is associated with a set of sites and can be used to generate an execution plan for the given query by keeping the fragmented relations in the subset fragmented at the sites where they are situated while replicating the other referenced relations at each of the processing sites. Among the alternatives, our algorithm picks the plan that gives the minimum response time. Our experimental results show that our algorithm improves response time significantly.

1 Introduction

Query processing in distributed database systems has been an active research area for many years. Many algorithms [1, 3, 4, 12, 14, 15, 19, 20] have been proposed. Some of the algorithms [4, 6, 13, 18, 20] take advantage of fragmentation to process queries. For example, the algorithm given in [20], called Fragment and Replicate Strategy (FRS) algorithm, is based on the same “fragment and replicate” principle as that of distributed INGRES[17, 18]. However, it takes into account not only the amount of data transferred and processed at individual sites but also the presence or absence of fast access paths (e.g., indices) to reduce response time. It also considers situations where fragments have duplicate copies, different sites have different processing speeds and so on. The main feature of FRS is to allow parallel processing. The algorithm chooses one relation to remain fragmented at the sites where they are situated while replicating the other relations at those sites. The query is decomposed into the same number of subqueries as the number of sites and each subquery is processed at one of these sites.

In this paper, we develop a new strategy to improve the FRS algorithm proposed in [20]. Our algorithm chooses more than one relation to remain fragmented at the sites where they are situated while replicating the other relations at those sites. Thus, the communication costs and local processing costs can both be reduced and the response time of queries can be improved.

The rest of this paper is organized as follows. In section 2, we give brief descriptions of our database environment and the query processing strategy. In section 3, we describe our new query processing strategy. The performance of our strategy is discussed in section 4. Finally, in section 5, we conclude the paper with the directions of future research.

2 Environment and Query Processing Strategy

Our environment is based on the concept of shared-nothing [16] architecture in which processors do not share disk drives and random access memory. The sites over which a database is distributed are connected in a local area network (Ethernet). Each site is managed by a local DBMS. A given query is submitted to a front end system that parses the query and determines a strategy to process the query[11]. The strategy usually consists of decomposing the query into several subqueries and assigning them to the local DBMSs at different sites. The front end system then waits for all the subqueries to finish executing. The partial results from the processing sites are then combined to produce the final result.

In this paper, we assume that all queries are of the form

$$Q = \{ \textit{target} \mid \textit{qualification} \}$$

where *target* is the list of projected attributes to be output to the user, *qualification* is the list of equijoin attributes. An attribute in a given query Q is called a *joining attribute* if it is included in the qualification part of the query.

A relation may be either horizontally fragmented or unfragmented. Both fragmented and unfragmented relations may have duplicate copies at several sites.

Definition 1 A *partition* of a relation R_i is a set of fragments $\{F_{ij}\}$, $1 \leq j \leq k_i$, with $F_{ij} \cap F_{it} = \emptyset$, for $j \neq t$ and $R_i = \cup_j F_{ij}$. ■

Definition 2 Let an attribute A of relation R_1 and an attribute B of relation R_2 be defined on the same domain. Assume $\{F_{1j}\}$, $1 \leq j \leq k$, be a partition of R_1 . If R_2 is partitioned into k fragments, F_{2j} , $1 \leq j \leq k$, such that $R_1 \bowtie_{A=B} R_2 = \cup_j F_{1j} \bowtie_{A=B} F_{2j}$, then we say R_2 has *partition dependency* on R_1 . ■

One should note that the above definition is different from key range partitioning or hash partitioning

[7, 10] where both relations are independently partitioned based on the values of a given attribute. In our definition, the partition of R_2 is dependent on that of R_1 . Usually, the partitioning of R_1 depends on the semantics of the application. For example, in a company, the employee table E could be partitioned based on the departments in which the employees work. If the dependent table D is partitioned into the same number of fragments as the employee table such that all the dependents of the employees belonging to fragment E_i are in fragment D_i , then relation D has partition dependency on relation E .

In this paper, we assume that at least one fragmented relation is referenced by the query and some fragments of referenced relations are replicated in some sites. For example, employee records for some departments may be replicated for higher reliability and/or availability while those for the other departments have a single copy. The replication decision is made in the initial data placement that could have been influenced by various factors, including available computer resources, importance of data, frequencies of access, and etc. We use CS_{ij} to denote the set of sites that contain a copy of fragment F_{ij} ; CS_u to denote the set of sites that contain a copy of unfragmented relation R_u ; $F_{ij,d}$ to denote the physical copy of fragment F_{ij} of relation R_i at site d , $d \in CS_{ij}$. We assume that no two copies of different fragments of a fragmented relation are situated at the same site; that is $CS_{ij} \cap CS_{ik} = \emptyset$, if $j \neq k$.

Definition 3 A *cover* of relation R_i , $\text{COVER}(R_i)$, is a partition of relation R_i with each fragment F_{ij} in the partition replaced by exactly one physical copy of the fragment $F_{ij,d}$, $d \in CS_{ij}$. ■

Definition 4 Let $\{F_{1jd}\}$ and $\{F_{2jd'}\}$, $1 \leq j \leq k$, be the covers of R_1 and R_2 , respectively. There is a *placement dependency* between these two covers on attribute A if there is a partition dependency between relations R_1 and R_2 on attribute A (R_1 have partition dependency on R_2 or the reverse) and for every j , $1 \leq j \leq k$, the fragments F_{1jd} and $F_{2jd'}$ are in the same site, i.e. $d = d'$. ■

Proposition: If there is placement dependency between a cover of $R_1(\{F_{1jd}\}, 1 \leq j \leq k)$ and a cover of $R_2(\{F_{2jd'}\}, 1 \leq j \leq k)$ on attribute A , then $R_1 \bowtie_A R_2$ can be decomposed into k subqueries, each of which can be processed at a site locally without data transfer. The union of the results of the subqueries forms the answer to $R_1 \bowtie_A R_2$. ■

An algorithm making use of the above basic idea to process distributed queries was proposed in [20]. For a given query, the locally processable query (LPQ) algorithm first decides whether the query can be processed locally without data transfer. If the query is locally processable, the algorithm decides a set of processing sites. The original query is decomposed into a number of subqueries, one for each processing site. If the query is not locally processable, the FRS algorithm is used to choose a strategy to process the query. For each

fragmented relation, FRS algorithm estimates the response time if that relation is left fragmented and all the other relations are replicated at each of the processing sites. The given query can also be processed by replicating all the referenced relations at a single site. The strategy that has the minimum estimated response time is chosen. Now let us use the following example to illustrate how the algorithm works.

Example 1 Let a query be

$Q = \{R_1.A, R_2.C \mid R_1.A = R_2.A \wedge R_2.B = R_3.C\}$ which references fragmented relations R_1 and R_2 and an unfragmented relation R_3 . There is a partition dependency between R_1 and R_2 on their join attribute A . Initial data distribution are the following: $CS_{11} = \{1, 2\}$, $CS_{12} = \{3\}$, $CS_{21} = \{1\}$, $CS_{22} = \{3\}$ and $CS_3 = \{1\}$.

First, the LPQ algorithm is applied to recognize whether Q is locally processable. Since $CS_{12} \cap CS_3 = \emptyset$ and $CS_{22} \cap CS_3 = \emptyset$, Q is nonlocally processable. Thus, the FRS algorithm is used to choose a strategy to process Q . The FRS algorithm performs the following three steps:

Step 1: find the minimum response time of the strategies using only one site (site 1, 2 or 3) to process Q .

If Q is processed only at site 1, the response time is the sum of the time taken to bring the remote fragments F_{12} and F_{22} to site 1, the time to do the unions, and the time to do the join operations $(F_{11} \cup F_{12}) \bowtie (F_{21} \cup F_{22}) \bowtie R_3$. Similarly, the response time of processing only at site 2 or site 3 can be obtained. The one with minimum response time among the three is chosen.

Step 2: determine the relation to remain fragmented and the set of processing sites.

(1) If R_1 is left fragmented, two sets of sites, $\{1, 3\}$ and $\{2, 3\}$, can be chosen to process Q . If Q is processed at sites 1 and 3, site 1 will need to get F_{22} from site 3 and site 3 will need to get F_{21} and R_3 from site 1, so that R_2 and R_3 are replicated at sites 1 and 3. Now, Q is decomposed into two subqueries:

$Q_1: \{F_{11}.A, R_2.C \mid F_{11}.A = R_2.A \wedge R_2.B = R_3.C\};$
 $Q_2: \{F_{12}.A, R_2.C \mid F_{12}.A = R_2.A \wedge R_2.B = R_3.C\}.$

where R_2 is the union of F_{21} and F_{22} . The final result of Q is the union of results of Q_1 and Q_2 . Since these two sites work in parallel, the response time will be the maximum of the total times spent at these two sites. Similarly, we can get the response time for processing Q at sites 2 and 3.

(2) If R_2 is left fragmented, Q will be processed at sites 1 and 3. Similarly the response time can be estimated.

Step 3: strategies obtained from *Step 1* and *Step 2* are compared, the one with the minimum response time is chosen. ■

However, since only one fragmented relation is chosen to remain fragmented in all strategies obtained in *Step 2* of the FRS algorithm, the algorithm has certain drawbacks in terms of efficiency and response time under some circumstances. For the above example, because there is a partition dependency between R_1 and R_2 on attribute A , and $CS_{11} \cap CS_{21} = \{1\}$ and $CS_{12} \cap CS_{22} = \{3\}$, we can find two covers $\text{COVER}(R_1) = (F_{111}, F_{123})$ and $\text{COVER}(R_2) =$

(F_{211}, F_{223}) such that there is a placement dependency between these two covers on attribute A . Therefore, the join between R_1 and R_2 on attribute A can be locally processed without data transfer by using these two covers. Thus, we can keep both R_1 and R_2 fragmented at sites 1 and 3 and only replicate relation R_3 to site 3. As a result, Q is decomposed into two subqueries:

$$Q_1: \{F_{11}.A, F_{21}.C \mid F_{11}.A = F_{21}.A \wedge F_{21}.B = R_3.C\};$$

$$Q_2: \{F_{12}.A, F_{22}.C \mid F_{12}.A = F_{22}.A \wedge F_{22}.B = R_3.C\}.$$

It is obvious that the above strategy reduces the data transfer cost since only R_3 need to be transferred from site 1 to site 3. The local processing cost is also reduced because the subqueries only use a fragment of R_1 and R_2 instead of the whole relation. Furthermore, the time needed to reconstruct R_1 or R_2 (union the two fragments of R_1 or R_2) is eliminated too. As a result, the response time of the query can be improved. The above example clearly indicates that if placement dependency information is used, more than one relation can remain fragmented and response times can be improved.

In the next section, we provide an algorithm that uses placement dependency information optimally to minimize response times of queries. The algorithm consists of the following three steps:

- (1). For a given query, find all sets of relations such that the join operation(s) associated with them can be locally processed without data transfer and find corresponding sets of processing sites.
- (2). One of the above sets of relations is chosen and the corresponding set of processing sites is determined such that the response time is minimized.
- (3). Compare above strategy with the strategy using only one site to process the query, the one with minimum response time is chosen.

3 An Integrated Algorithm

Let SP contain the names of the n relations referenced by the query. We assume that there are f ($f \neq 0$) fragmented relations in SP , and every fragmented relation R_i in SP has k_i distinct fragments.

3.1 All Sets of Locally Processable Joins

Let LP and $LPCS$ be two sets whose members have a 1-to-1 correspondence. Each member of LP is a set of relations such that the join operation(s) associated with them can be locally processed without data transfer. The corresponding member of $LPCS$ is the set of processing sites.

For example, if there is a placement dependency between a cover of R_i (F_{i1p}, F_{i2q}) and a cover of R_j (F_{j1p}, F_{j2q}) on their join attribute, then one element of LP , say LP_k , contains relations R_i and R_j , i.e., $LP_k = \{R_i, R_j\}$; and its corresponding element $LPCS_k$ of $LPCS$ contains the processing sites p and q , i.e., $LPCS_k = (p, q)$.

In order to get LP and $LPCS$, we first consider the fragmented relations in SP . The fragmented relations that have placement dependency between their covers

become an element of LP , and the sites in these covers become a corresponding element of $LPCS$. If some fragmented relations have placement dependency on several different covers, we use different elements of LP and $LPCS$ to store them. For example, if fragmented relations R_i and R_j have placement dependencies between covers (F_{i1p}, F_{i2q}) and (F_{j1p}, F_{j2q}), and between covers (F_{i1p}, F_{i2r}) and (F_{j1p}, F_{j2r}), then one element of LP , say LP_k , contains R_i and R_j , and the corresponding member of $LPCS$, $LPCS_k$, contains (p, q) . Another element of LP , say LP_t , also contains R_i and R_j , but the corresponding member of $LPCS$, $LPCS_t$, contains (p, r) .

For an unfragmented relation R_u , if there is a join operation between R_u and a relation in an element of LP and the set of sites having copies of R_u is a superset of the set of sites in the corresponding element of $LPCS$, we add R_u into the element of LP . For example, let R_u be an unfragmented relation and $CS_u = \{p, q\}$, and $LP_k = \{R_i, R_j\}$ in LP and $LPCS_k = \{p, q\}$ in $LPCS$. If there is a join operation between R_u and R_i and/or between R_u and R_j , the relation R_u will be added into element LP_k , that is $LP_k = \{R_i, R_j, R_u\}$. The join operations associated with R_i , R_j and R_u can be processed at sites p and q without data transfer.

In the following subsections, we discuss the construction of LP and $LPCS$.

3.1.1 Data Structure

We use record REC to store names of relations that are referenced by the query. Every REC record contains at least one fragmented relation. The fragmented relations that have placement dependency between their covers belong to the same record. The join operation(s) associated with the relations that belong to the same record can be processed locally without data transfer.

All REC records are linked to some nodes of a rooted tree T . Each node of T except the root that is dummy corresponds to a site. A path from the root to a node d is a list of sites denoted as $path(d)$. If a REC record is linked to d , then for each fragmented relation in the REC record, its fragments at the sites of $path(d)$ form a cover of the relation; for each unfragmented relation in the REC record, there exist a copy at every site in $path(d)$. For example, in Figure 1, record REC_p containing fragmented relations R_i and R_j is linked to node 5 in the path (2, 4, 5). This means that 1) R_i has a cover ($F_{i12}, F_{i24}, F_{i35}$) and R_j has a cover ($F_{j12}, F_{j24}, F_{j35}$), and 2) the join operation between R_i and R_j can be processed at sites 2, 4 and 5 without data transfer. For node 3 in the path (2, 1, 3), it links record REC_q containing fragmented relation R_i and unfragmented relation R_u . This means that 1) R_i has a cover ($F_{i12}, F_{i21}, F_{i33}$) and there is a copy of R_u at sites 2, 1 and 3 respectively, and 2) the join operation between R_i and R_u can be processed at sites 2, 1 and 3 without data transfer.

The set of relations in every REC record will become an element of LP and the sites in the corre-

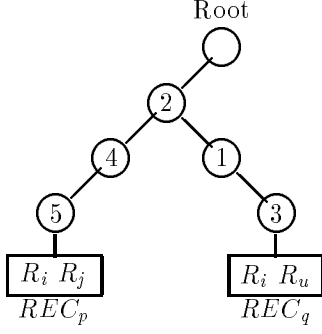


Figure 1: Tree T

sponding path will become a corresponding element of $LPCS$.

3.1.2 Fragmented Relations

For a fragmented relation R_i , which has k_i (let $m = k_i$) fragments, we get its cover by using CS_{ij} . For a cover of R_i , say $(F_{i1d_1}, F_{i2d_2}, \dots, F_{imd_m})$, $d_j \in CS_{ij}$, $1 \leq j \leq m$, sites d_1, d_2, \dots, d_m in that order forms a path $p = (d_1, d_2, \dots, d_m)$. We search tree T and consider the following three cases:

Case 1: p is not in T . We create path p and link a new REC record containing R_i to node d_m in the path.

Case 2: p exists in T and there is no REC record linked to node d_m in p . We simply create a new REC record containing R_i and link it to node d_m in the path.

Case 3: p exists in T and there is at least one REC record linked to node d_m in the path. We check every REC record that is linked to node d_m and do the following:

case 3.1 For every REC record that is linked to node d_m , there is no join operation and/or partition dependency on the join attribute(s) between R_i and any relation in the record. It means that there is no placement dependency between the cover of R_i and the cover of any relation in every REC record linked to node d_m , though they have covers with the same sites order. Therefore, we create a new REC record containing R_i and link it to node d_m .

case 3.2 For a record, say REC_k , that is linked to node d_m , there is a join operation and partition dependency on the join attribute(s) between R_i and a relation, say R_j , belonging to the record. This means that there is a placement dependency between the cover of R_i and the cover of R_j . Thus, we add R_i to REC_k .

If two or more records linked to the same node have a common relation, these records will be combined to form a single REC record.

Example 2 Let R_a , R_b and R_c be three fragmented relations and (F_{a1q}, F_{a2r}) , (F_{b1q}, F_{b2r}) and (F_{c1q}, F_{c2r}) be their covers respectively. Assume that the query is $R_a \bowtie R_c \bowtie R_b$ and there is partition dependency

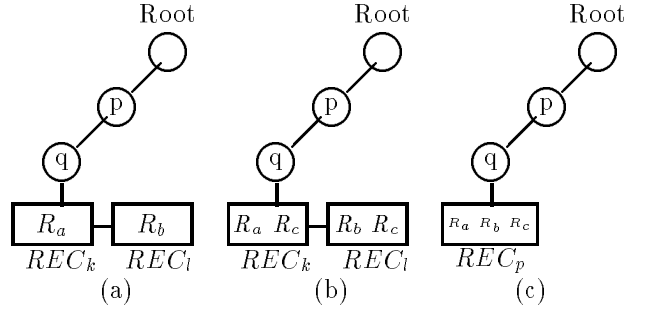


Figure 2: Tree T

between R_a and R_c , and between R_b and R_c on their join attributes, respectively.

Initially, the path (q, r) corresponding to the cover of R_a does not exist. We create the path and REC_k as shown in Figure 2(a). For R_b , case 3.1 arises, we create REC_l as shown in Figure 2(a). When R_c is considered, we have case 3.2. Thus, we add R_c to both REC_k and REC_l as shown in Figure 2(b). Since R_c belongs to both REC_k and REC_l that are linked to node q , the join operations associated with R_a , R_b and R_c can be locally processed at sites q and r . Therefore, we combine REC_k and REC_l into a single record REC_p containing R_a , R_b and R_c as shown in Figure 2(c). ■

The above procedure is repeated for every fragmented relation in SP .

3.1.3 Unfragmented Relations

For every unfragmented relation R_{f+1} to R_n in SP , we compare it with the relation(s) belonging to the REC record(s) in the tree T to find whether the join operation associated with them can be locally processed. For an unfragmented relation, say R_u , and a relation, say R_i , belonging to a REC record, say REC_k , if 1) there is a join operation between R_u and R_i , and 2) there is a copy of R_u at each site within the path from the root node to the node that links the REC_k record, then the join operation between R_u and R_i can be locally processed without data transfer. Thus, we add R_u to REC_k .

3.1.4 Get LP and $LPCS$

For a REC record in T , say REC_k , if it contains only one relation, say R_i , we consider the following cases:

Case 1: If R_i also belongs to other REC records that contain more than one relation, then there is at least one relation whose join operation with R_i can be locally processed. In this case, we take a heuristic approach to reduce query optimization time by eliminating the strategy corresponding to REC_k from consideration.

Case 2: Otherwise, we keep REC_k in T .

If every REC record contains only one fragmented relation, it means that no more than one fragmented relation can remain fragmented and no join operation

between any two fragmented relations can be locally processed. Under this special situation, our strategy is the same as the FRS[20].

Now we can get *LP* and *LPCS*. For every *REC* record in *T*, the set of relations in the record becomes an element of *LP*, and the sites in the path from the root node to the node linking the record becomes a corresponding element of *LPCS*.

3.1.5 The Algorithm

Algorithm FLPSETS: Find *LP* and *LPCS*.

Input: *SP*, *CS*, partition dependency,
join operation information.

Output: *LP* and *LPCS*.

(1) processing fragmented relations in *SP*:

```

create root record of T; /* initialize T */
FOR i = 1 TO f
{
  For each cover of Ri
  {
    P ← the path corresponding to the cover;
    /* path(T) is the set of all paths in T */
    IF (P ∈ path(T)) THEN
    {
      For each REC record linked to P
      {
        IF (there is a partition dependency
            between Ri and a relation in the REC
            record on their join attribute(s)) THEN
          add relation Ri to the REC record;
      }
      combine the REC records belonging to P
      and containing Ri into one record;
    }
    ELSE
    {
      create the path P in T;
    }
    IF (Ri has not been added to any REC), THEN
      create a REC record containing Ri
      and link it to P;
  }
}
(2) processing unfragmented relations in SP:
FOR u = f + 1 TO n
{
  For every REC record in T
  {
    IF ((there is a join operation between Ru
        and a relation in the REC record) AND
        (the set of sites in the corresponding path
        is a subset of CSu)) THEN
      add Ru to that REC record;
  }
}
(3) get LP and LPCS:
For every REC record in T
{
  IF (REC contains more than one relation
  OR the relation is not a proper subset

```

```

of any other record in T) THEN
  add an element of LP containing the
  relations of REC and an element of
  LPCS containing the path from root to REC
}

```

3.2 Determine the Strategy with Minimum Response Time

LP and *LPCS* have the same number of elements. Any corresponding elements of *LP* and *LPCS*, say *LP_k* and *LPCS_k*, can be used to form a processing strategy. *LPCS_k* contains the set of processing sites and *LP_k* is the set of relations that can remain fragmented. The referenced relations that are not in *LP_k* will be replicated at each processing site. In addition, we also consider the simple strategy of single site processing, i.e., replicate all the referenced relations at a single site and process the query at that site. Finally, all the strategies are compared and the one with minimum response time is chosen.

3.2.1 Cost Model

To estimate the response times of the strategies, it is necessary to estimate the local processing (join) cost and data transmission cost.

We use a linear cost model for joins. Particularly, for relations *R₁* and *R₂* having sizes *X₁* and *X₂* respectively, the cost of join

$$C(X_1, X_2) = \sum_{i=1}^2 h(X_i),$$

where *h()* is a linear function whose parameters depend on the availability of fast access path, i.e.,

$$h(X_i) = \begin{cases} f(X_i) & \text{if a fast access path exists for} \\ & \text{the join operation;} \\ n(X_i) & \text{otherwise;} \end{cases}$$

When no confusion arises, we shall replace the size of the relation or fragment by its name. If *R₁* and *R₂* are fragmented with fragments *F₁₁*, ..., *F_{1n}* and *F₂₁*, ..., *F_{2m}*, then the cost is assumed to be *h(F₁₁)* + ... + *h(F_{1n})* + *h(F₂₁)* + ... + *h(F_{2m})*. This is clearly an approximation. The reasons for choosing such a simplified cost function are as follows. The analysis given in [8] has demonstrated that a linear cost model is sufficiently robust to cover the usual join algorithms, including nested-loop join (with an indexed inner relation or without an index), hash join, and merge join. Note that sort-merge join is not linear in the cardinalities of the input relations. However, most systems do not use sort-merge join, since in situations where merge join requires sorting of the input, either hash join or nested-loop join is almost always preferable to sort-merge[9]. Linear cost models have also been evaluated experimentally for nested-loop and merge joins, and were found to be relatively accurate of the performance of a variety of commercial systems [5].

We further make the assumption that when a relation/fragment is sent from one site to another, its fast access path, if one exists in the sending site, will not be available in the receiving site. Then the cost

of processing the relation/fragment X is $n(X)$ at the destination site.

Clearly, all required data needed by a site have to be received before processing can take place at the site. We use $t(X)$ to denote the cost to transmit a relation/fragment whose size is X . The total cost at a site is the cost the site will be able to finish the part of the query assigned to the site for processing.

3.2.2 Relations to Remain Fragmented

Assume the fragmented relation(s) in $LP_k \in LP$ be chosen to remain fragmented and the sites in $LPCS_k$ be used as processing sites. For a given LP_k and $LPCS_k$, the join(s) associated with the relations in LP_k can be locally processed at sites belonging to $LPCS_k$ without data transfer.

Local Processing Cost. Suppose a site in $LPCS_k$, say j , is used as a processing site. The cost for processing all fragmented relations in LP_k at site j is $PWF_1(j) = \sum_{i=1, R_i \in LP_k}^f h(X_i)$, where X_i is the fragment/relation of $R_i \in LP_k$ at site j and $h(X_i)$ is the cost function for the join operation that we described in the previous subsection.

For a fragmented relation $R_i \notin LP_k$, if there is no any fragment of R_i existing in site j , the cost for processing R_i at site j is $n(R_i)$. If there is a copy of a fragment of R_i , say X_i , at site j , the fast access path, if any, could be utilized to reduce the cost, and the saving is $n(X_i) - h(X_i)$. Thus, for R_i , the cost at site j is $n(R_i) - (n(X_i) - h(X_i))$. Therefore, the cost for processing fragmented relations, not in LP_k , accessed by the query at site j is $PWF_2(j) = \sum_{i=1, R_i \notin LP_k}^f n(R_i) - (n(X_i) - h(X_i))$, where X_i is a fragment of R_i that initially exists at site j .

Thus, the total cost for processing all fragmented relations at site j is $PWF(j) = PWF_1(j) + PWF_2(j)$.

For an unfragmented relation R_u that is to be processed at site j , the cost at site j is $h(R_u)$ or $n(R_u)$ depending on whether there is a copy of R_u at site j . The total cost for processing unfragmented relations accessed by the query at site j is $PWU(j) = \sum_{u=f+1}^n g(R_u)$, where function $g(R_u)$ is either $h(R_u)$ or $n(R_u)$.

The total cost for local processing at site j is $PWF(j) + PWU(j)$.

Communication Cost. All required data needed by a site have to be received before processing can take place at the site. If a site, say site j , is used to process the query, all distinct data entities of relations not in LP_k except those initially there must be transmitted to site j . Thus, the total cost to transfer the data to site j is $TW(j) = \sum_{i=1, R_i \notin LP_k}^n t(R_i) - t(X_i)$, where X_i is the fragment of relation R_i initially existing at site j .

The Set of Relations with Minimum Response Time. Let $T(k, j)$ be the total cost at site $j \in LPCS_k$

for the strategy of keeping the relations in LP_k fragmented. Then, $T(k, j) = PWF(j) + PWU(j) + TW(j)$. Since there are more than one processing site in $LPCS_k$, the response time of $LPCS_k$ is the maximum cost among all processing sites in $LPCS_k$. If $LP_k \in LP$ is chosen, the response time is

$$T_{cost}(k) = \max_{j \in LPCS_k} \{T(k, j)\}.$$

At last, we choose a set of relations in an element of LP with minimum response time among all elements in LP and $R_{cost} = \min_{LP_k \in LP} \{T_{cost}(k)\}$.

3.2.3 The Minimum Response Time Strategy

Finally, R_{cost} is compared to the cost of single site processing, i.e., replicate all referenced relations at a single site and process the original query at that site. Each candidate site is considered. The strategy having the minimum cost is chosen. If a strategy using single site processing is chosen, the processing site is given. Otherwise, a set of relations (an element of LP) will remain fragmented and the corresponding element of $LPCS$ gives the set of processing sites.

3.2.4 Algorithm FPRS

Algorithm FPRS: Find the strategy with minimum response time.

Input: SP, CS, LP and $LPCS$.

Output: the set of relations to remain fragmented and the set processing sites.

(1) relations to remain fragmented:

$R_{cost} \leftarrow \infty;$

$x \leftarrow 0;$

For each $LP_k \in LP$

{

For each site $j \in LPCS_k$, calculate the total cost:

$T(k, j) = PWF(j) + PWU(j) + TW(j);$

compute the response time:

$T_{cost}(k) = \max_{j \in LPCS_k} \{T(k, j)\};$

IF $(T_{cost}(k) \leq R_{cost})$ THEN

{

$R_{cost} \leftarrow T_{cost}(k);$

$x \leftarrow k;$

}

}

(2) single site processing strategy (SSP):

$T_{ssp} \leftarrow \infty;$ /* response time of SSP */

$S_{ssp} \leftarrow \text{Null};$ /* site of SSP */

For each site j

{

IF $(T_{ssp}(j) \leq T_{ssp})$ THEN

{

$T_{ssp} \leftarrow T_{ssp}(j);$

$S_{ssp} \leftarrow j;$

}

}

(3) The minimum response time strategy:

IF $(T_{ssp} \geq R_{cost})$ THEN

return LP_x and $LPCS_x;$

/* keep the relations in LP_x fragmented and */

```

/* replicate the others at each site in  $LPCS_x$  */
ELSE
return {Null} and  $\{S_{ssp}\}$ .

```

4 Performance of the Algorithm

In order to study the performance of our query processing strategy and compare to the strategy studied in [11, 20], the following simulation is conducted.

Based on the benchmark results in [2] and our practical experience, the following cost functions for the join operation is used. $f(X) = \frac{1}{4} * |X|$ and $n(X) = |X|$, where $|X|$ is the size of the data entity X . We also use cost function $t(X) = |X|$ for transferring a relation/fragment, where $|X|$ is the size of the relation/fragment X .

The dimension of the initial data distribution for each experiment is $m * n$, where m , which changes from 3 to 10, is the number of relations accessed by the query, and n , which changes from 2 to 8, is the number of sites accessed by the query. For each experiment, a combination of m and n , 200 cases are randomly generated, tested and the average result is recorded.

A test case is generated by giving an $m * n$ matrix, which represents the initial data fragmentation, distribution, and sizes of the fragments and relations. Each entry of the matrix contains some information such as whether the entity is a relation or a fragment, whether or not it is a copy of some other data entity, its size, and whether or not it has a fast access path. There are 200 arbitrarily generated initial data distribution matrices of each size. In these matrices, the number of fragmented relations ranges from 1 to m , the number of distinct fragments of a fragmented relation ranges from 2 to 4, the number of copies of a fragment ranges from 1 to 3, and the number of copies of unfragmented relation ranges from 1 to 5. The range of sizes of relations is from 1K tuples to 20K tuples. There are 10 different sets of arbitrarily selected sizes. The 200 test cases of each experiment are generated by combining each of the 200 matrices with one set of sizes of relations.

In order to compare our strategy to the original strategy, we define the following two performance metrics:

- (1) Favorability: the *favorability* of our strategy to the original strategy is defined as $(N/T) * 100\%$, where T is the total number of test cases and N is the number of test cases that our strategy yields less response time than the original strategy.
- (2) Improvement: the improvement of our strategy to the original strategy is defined as $((Y_f - Y_n)/Y_f) * 100\%$, where Y_f and Y_n are the response times obtained by applying the original strategy and our strategy respectively.

The experimental results are shown in Tables 1 and 2. From Table 1, one can easily notice that when the number of sites is fixed, the favorability of our strategy increases as the number of relations increases (columns of Table 1). This is as expected since

m	Number of Sites							A
	2	3	4	5	6	7	8	
3	26.5	9.0	10.0	15.5	6.5	10.5	8.0	12.3
4	50.0	28.5	24.5	19.5	17.5	17.5	18.0	25.1
5	54.0	45.0	31.0	27.5	26.0	17.5	22.5	31.9
6	60.5	44.5	34.5	27.0	20.0	20.0	21.5	32.6
7	70.0	50.0	41.5	29.5	31.5	23.5	26.5	38.9
8	71.5	54.5	45.0	34.0	34.5	29.5	25.5	42.1
9	73.5	56.0	50.0	45.5	36.5	25.5	33.0	45.7
10	78.5	61.0	49.5	39.5	36.0	30.0	33.5	46.9
A	60.6	43.6	35.8	29.8	26.1	21.8	23.6	

m – number of relations, A – average

Table 1: Favorability of Our Strategy $((N/T)*100\%)$

m	Number of Sites							A
	2	3	4	5	6	7	8	
3	27.3	27.3	31.8	23.1	27.9	19.4	23.3	25.7
4	26.7	18.6	21.0	21.1	17.7	18.0	22.8	20.8
5	30.1	15.4	17.3	17.9	15.5	16.9	13.9	18.1
6	28.0	13.8	17.1	15.3	17.5	14.4	15.8	17.4
7	26.9	12.6	13.6	15.0	15.8	11.7	12.3	15.4
8	28.2	12.0	13.3	14.0	13.4	10.9	10.4	14.6
9	29.9	13.0	14.0	13.4	13.3	12.0	10.0	15.1
10	30.3	13.8	13.3	13.2	13.3	12.1	11.3	15.3
A	28.4	15.8	17.7	16.6	16.8	14.4	15.0	

m – number of relations, A – average

Table 2: Improvement Factor $((Y_f - Y_n)/Y_f) * 100\%$

the favorability of our algorithm is actually the probability that part of the referenced relations have placement dependency. Recall that when the referenced relations have no placement dependency at all or have fully placement dependency (i.e. all the referenced relations can remain fragmented), our algorithm generate the same execution plan as that of [20]. When the number of relations increases, the probability that some referenced relations have placement dependency increases. As a result, the favorability of our strategy increases. However, when the number of sites increases, the data distribution becomes sparse and the chances that the corresponding fragments of the referenced relations are placed at the same site become smaller. As a consequence, our algorithm becomes less favorable as the number of sites increases. This is clearly reflected in Table 1. However, we should point out that the data distribution is generated randomly in our experiments while in practise, the corresponding fragments of relations are likely to be placed at the same site. For example, information of dependents of an employee are stored at the site where the information of the employee is stored. If this is the case, higher favorability of our algorithm can be expected.

As shown in Table 2, the average improvement decreases as the number of relations increases. In our algorithm, only the fragmented relations that have placement dependency between their covers can re-

main fragmented and the gains in local processing cost and transmission cost are due mainly to these relations. When the number of referenced relations increases, the expected percentage of referenced relations having placement dependency also decreases. As a result, the gains in local processing cost and transmission cost relative to the total cost decreases. Thus, the average improvement decreases as the number of referenced relations increases though the decrease is rather slow. We also examine the impact of the number of sites on the improvement. The experimental results show that when the number of sites with uniform processing speed increases, the improvement does not increase or decrease monotonically but fluctuate around 16%.

5 Conclusion

We present an algorithm for processing distributed queries by using the placement dependency information. Since more than one fragmented relation can remain fragmented with our algorithm under significant number of cases, the cost of data transfer and query processing is less than that of the algorithm given in [20]. If there is no placement dependency between covers of fragmented relations referenced by the query, our algorithm chooses one fragmented relation to remain fragmented. Under this case, the cost of using our algorithm is the same as that of using the algorithm given in [20]. The algorithms given in [20] are special cases of our algorithm. In the worst case, our algorithm has the same performance as that of the algorithm given in [20]. However, when only some of the referenced relations have placement dependency, our algorithm outperforms that of [20]. Our experimental results show that our algorithm can improve performance significantly for substantial number of cases.

As shown in the previous sections, the cost of assembling results of the subqueries executing in the processing sites is not included in our cost model. When a partial result is transferred to the site where the query is issued, it is to be unioned with the partial results that have been assembled so far. The cost of performing the union may depend on the number of processing results. We intend to explore this issue in an implementation in the near future.

References

- [1] Apers, P., Hevner, A. and Yao, S. B.: Optimization Algorithm for Distributed Queries. *IEEE Trans. on Soft. Engr.*, Vol. 9, No. 1, pp. 57-68, Jan. 1983
- [2] Bitton, D. and D. J. DeWitt and C. Turbyfill: Benchmarking Database Systems – A Systematic Approach. *Proc. of the 9th Int'l Conf. on Very Large Data Bases (VLDB)*, pp. 8-19, Florence, Italy, Oct. 1983.
- [3] Chang, J. M.: A Heuristic Approach to Distributed Query Processing. *Proc. 8th Int'l Conf. on VLDB*, pp. 54-61, Mexico City, Aug. 1982.
- [4] Chen, A. L. P. and Li, V. O. K.: Optimizing Joins in Fragmented Database Systems on a Broadcast Local Network. *IEEE Trans. on Soft. Engr.*, Vol. 15, No. 1, pp. 26-38, Jan. 1989.
- [5] Du, W., Krishnamurthy, R. and Shan, M. C.: Query Optimization in Heterogeneous DBMS. *Proc. of the 18th Int'l Conf. on VLDB*, Vancouver, Aug. 1992.
- [6] Gavish, B. and Segev, A.: Set Query Optimization in Distributed Database Systems. *ACM TODS*, Vol. 11, No. 3, 1986.
- [7] Ghandeharizadeh, S. and DeWitt, D. J.: A Multiuser Performance Analysis of Alternative Declustering Strategies. *Proc. 6th IEEE Int'l Conf. on Data Engr.*, pp. 466-475, Los Angeles, CA, Feb. 1990.
- [8] Hellerstein, J. M.: Practical Predicate Placement. *Proc. 1994 ACM SIGMOD Conf.*, pp. 325-335, Minneapolis, Minnesota, May 1994.
- [9] Hellerstein, J. M. and Stonebraker, M.: Predicate Migration: Optimizing Queries with Expensive Predicates. *Proc. 1993 ACM SIGMOD Conf.*, pp. 267-276, Washington, DC, May 1993.
- [10] Hsiao, H. I. and DeWitt D. J.: Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. *Proc. 6th IEEE Int'l Conf. on Data Engr.*, pp. 456-465, Los Angeles, CA, Feb. 1990.
- [11] Liu, C. and Yu, C.: Performance Issues in Distributed Query Processing. *IEEE Trans. on Par. and Distr. Sys.*, Vol. 4, No. 8, pp. 889-905, Aug. 1993.
- [12] Pramanik, S. and Vineyard, D.: Optimizing Join Queries in Distributed Databases. *IEEE Trans. on Soft. Engr.*, Vol. 14, No. 9, pp. 1319-1326, Sept. 1988.
- [13] Segev, A.: Optimization of Join Operations In Horizontal Partitioned Database System. *ACM TODS*, Vol. 11, No. 1, pp. 48-80, Mar. 1986.
- [14] Shasha, D. and Wang, T. L.: Optimizing Equijoin Queries in Distributed Databases where Relations Are Hash Partitioned. *ACM TODS*, Vol. 16, No. 2, pp. 279-308, June 1991.
- [15] Stamos, J. W. and Young H. C.: A Symmetric Fragment and Replicate Algorithm for Distributed Joins. *IEEE Trans. on Par. and Distr. Sys.*, Vol. 4, No. 12, pp. 1345-1354, Dec. 1993.
- [16] Stonebraker, M.: The case for shared-nothing. *Data Base Engineering*, Vol. 9, No. 1, pp. 4-9, Mar. 1986.
- [17] Stonebraker, M. and Neuhold, E: A Distributed Data Base Version of INGRES. *Second Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 19-36, 1977.
- [18] Wong, E., and Katz, R. H.: Distributing a Database for Parallelism. *Proc. 1983 ACM SIGMOD Conf.*, pp. 23-29, San Jose, CA, May 1983.
- [19] Yu, C. T., and Chang, C. C.: Distributed Query Processing. *ACM Computing Survey*, Vol. 16, No. 4, pp. 399-433, Dec. 1984.
- [20] Yu, C. T., Guh, K. C., Zhang, W., Templeton, M., Brill, D., Chen., A. L. P.: Algorithms to Process Distributed Queries in Fast Local Networks. *IEEE Trans. on Comp.*, Vol. 36, No. 10, pp. 1153-1164, Oct. 1987.