Query Processing and Optimization in Distributed Database Systems

B.M. Monjurul Alom, Frans Henskens and Michael Hannaford

School of Electrical Engineering. & Computer Science, University of Newcastle, AUSTRALIA

Summary
Query processing is an important concern in the field of distributed databases. The main problem is: if a query can be decomposed into subqueries that require operations at geographically separated databases, determine the sequence and the sites for performing this set of operations such that the operating cost (communication cost and processing cost) for processing this query is minimized. The problem is complicated by the fact that query processing not only depends on the operations of the query, but also on the parameter values associated with the query. Distributed query processing is an important factor in the overall performance of a distributed database system.

Query optimization is a difficult task in a distributed client/server environment as data location becomes a major factor. In order to optimize queries accurately, sufficient information must be available to determine which data access techniques are most effective (for example, table and column cardinality, organization information, and index availability). Optimization algorithms have an important impact on the performance of distributed query processing.

In this paper, we describe the distributed query optimization problem in detail. We then present a (ARRQ) technique to process queries with a minimum quantity of intersite data transfer. The technique can be used to process the query where all of the relations referenced by a query are non-fragmented but distributed in different sites. The proposed technique is used to determine which relations are to be partitioned into fragments, and where the fragments are to be sent for processing. The technique is efficient compared to other techniques, as it generally chooses more than one relation to remain fragmented which exploits parallelism, while replicating the other relations (excluding the fragmented relations) to the sites of the fragmented relations. Thus the communication costs and local processing costs can be reduced due to the reduced size of the fragmented relations and the response time of queries can be improved.

Key words:
Join, Semijoin, Query, FRS, PRS, LR, and Optimization.

1. Introduction

Distributed and parallel processing is an efficient way of improving the performance of Database Management Systems (DBMSs) and applications that manipulate large volumes of data [1]. A distributed database management system (DDBMS) supports the formation (creation) and maintenance of distributed databases, where data are stored at different sites connected through a network. An objective of a DDBMSs is to present an easy and unified interface to the users so that they can access the databases as if there were a single database [2]. Another important objective of DDBMS is to process distributed queries efficiently in addition to providing availability and reliability. Distributed database systems (DDBS) and distributed computing systems (DCS) differ in the resources to be shared. DCS share hard disks and printers etc. while DDBS distribute data, where the data as well as operations on the data items are equally important [3] .

Since data are geographically distributed in such a distributed relational database system, the processing of a distributed query is composed of the following three phases: local processing phase, reduction phase, and final processing phase [4]. The local processing phase involves local processing such as selections and projections; the reduction phase uses a sequence of reducers (i.e, semijoins and joins) to reduce the size of relations; and the final processing phase sends all resulting relations to the assembly site where the final result of the query is constructed. Clearly, a straightforward approach to processing a distributed query would be to send all relations directly to the assembly site, where all joins are performed. This naive method, however, is unfavourable due to its high transmission overhead and because little parallelism is exploited. In distributed query processing, partitioning a relation into fragments, union of the fragments to form a whole relation, and transferring a relation/fragment from one database to another database are common operations [2].

The optimizers of R* [5] and Distributed-INGRES [6] , take both local processing costs and communications costs into account. In R*, a join between two relations is performed at a single site by using the nested-loop method or the merge-scan method. For a general query, R* exhaustively enumerates all possible sequences of joins with all possible join methods and allocates joins at each possible site. Since each join is performed at only a single site, existence of multiple

Manuscript received September 5, 2009
Manuscript revised September 20, 2009

processors at different sites is not considered for improving performance through parallel execution. In contrast, Distributed-INGRES uses the "fragment and replicate" query processing strategy [7]. The strategy requires one of the relations referenced by a query to be fragmented and other relations to be replicated at the sites that have a fragment of the fragmented relation. 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. Its main feature is that it allows parallel processing.

Many other algorithms [8, 9], [10], [11], [12-15] also take advantage of fragmentation to process queries. For example, the algorithm given in [12] , called the Fragment and Replicate Strategy (FRS) algorithm, is based on the same principle as that of [7]. 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., indexes) to reduce response time. When all the relations referenced by a query are unfragmented, the FRS algorithm can not be used. Another strategy is to partition one of the referenced relations into a number of fragments and distribute the fragments to a number of sites so that the query can be processed in parallel at different sites [13] . In [15], a Partition and Replicate Strategy (PRS) algorithm is given to determine which relation and which copy of the relation is to be partitioned into fragments, how many fragments are to be produced, and where these fragments are to be sent for processing.

Both FRS and PRS require substantial data transfer and preparation before a query can be processed in parallel at different sites. By doing local reduction of the fragments or relations before transferring them, i.e., do some projections and selections, the data transfer cost can be reduced and subsequent local processing (join) cost may also be reduced due to smaller sized relations. However, performing local reduction takes time and delays data transfer. It is not always true that local reduction will reduce the response time for the processing of a given query. In order to reduce response time for a query, an algorithm is described in [14], called the Local Reduction (LR) algorithm which is used to decide the set of relations to be locally reduced before data transfer.

Two approaches namely semijoin [16] and join sequence [8] have been used to reduce the amount of data transmission required for the phases of distributed query processing. The semijoin operation from R_p to R_q, denoted by $R_q \ltimes R_p$, is defined as follows: Project R_p, on the join attribute of the join between R_p and R_q first, and then ship this projection to the site of R_q to remove nonmatching tuples from R_q. In addition to semijoins, join operations can also be used as reducers in processing distributed queries [17, 18] . Using join reducers, a query is translated into a sequence of joins, and each join is implemented locally by shipping one of the operand relations to the site of the other operand so as to exploit parallelism and minimize the processing overhead. Moreover, joins and semijoins can be combined to form an integrated scheme to further improve distributed query processing ([17] , [19]). As pointed out in [18] , the approach of combining join and semijoin operations can be more beneficial due to the inclusion of join reducers. Also, this approach can reduce the communication cost further by taking advantage of the removability of pure join attributes.

While a significant amount of research effort has been reported on developing algorithms based on joins and semijoins, there is relatively little progress made to the complexity of the distributed query processing. As a result, proving NP-hardness of, or devising polynomial-time algorithms for, certain distributed query optimization problems have been elaborated by many researchers [4]. However, due to their inherent difficulty, the complexity of the majority of problems on distributed query optimization remains unknown [4]. Traditionally, distributed query optimization techniques generate static query plans at compile time. The optimality of these plans depends on many parameters (such as the selectivities of operations, the transmission speeds and workloads of servers) that are not only difficult to estimate but are also often unpredictable and variable at runtime. The main difference between centralized and distributed query optimization is in the method-structure space module, which offers additional processing strategies and opportunities for transmitting data for processing at multiple sites. In [20], Ibaraki and Kameda describe query optimization as an NP-complete problem whilst using the nested loops join method.

In this paper we describe (ARRQ) a technique to process a query where all the relations referenced by a query are not fragmented but distributed in different sites. The proposed technique is used to determine which relations are to be partitioned into fragments, and where the fragments are to be sent for processing. Our objective is to process queries by exploiting parallelism, as well as minimizing the quantity of inter-site data transfer. The proposed technique provides better efficiency in terms of query processing cost when the given query references all the relations or more than one relation (for different sites) that presents (the where condition) predicates of the query. We are more concerned to fragment more than one referenced non fragmented relations as FRS is not applicable to processing distributed queries in which all of the relations which are non fragmented but referenced by a query. We also characterize distributed query optimization problems.

The remainder of this paper is organized as follows: different methodologies for distributed query processing are described in section 2. Components and problems of distributed query optimization are described in section 3. The proposed query processing technique is presented in section 4. The paper concludes with a discussion and final remarks in section 5.

2. Different Methodologies for Distributed Query Processing

P. Valduriez [1] has described a methodology of distributed query processing, and this is shown in Fig.1.

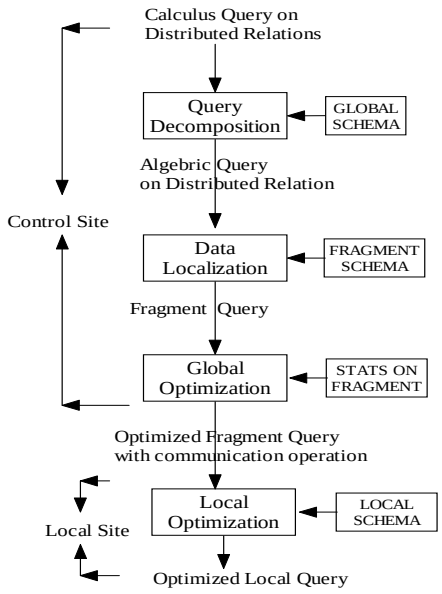

Fig. 1 Distributed Query Processing Methodology.

The input is a query on distributed data expressed in relational calculus. Four main layers are involved to map the distributed query into an optimized sequence of local operations, each acting on a local database. These layers perform the functions of query decomposition, data localization, global query optimization, and local query optimization. Query decomposition and data localization correspond to query rewriting. The first three layers are performed by a central site and use global information. Local optimization is done by the local sites.

The functionality of distributed query processing is demonstrated in the following examples using two different (semijoin and join) strategies:

Suppose a database is distributed into three different sites; for example Operation, Nursing, and ICU (Intensive Care Unit) sites. The schemas of these relations are R_{OP} (Patname, DOB, Admit, Discharge, Dept), R_{NU} (Patname, DOB, GP), and R_{ICU} (Patname, DOB, Weight, Type_work) respectively. It is required to find Patname, Admit, Type_work, and Weight from Operation site; wherein Dept should be Orthopedics, Admission must be before 1st January, Patname should be common between Operation and ICU sites, and the weights must be greater than seventy. The query in SQL is as follows:

Select Patname, Admit, Type_work, Weight
From R_{OP} , R_{ICU}
Where (R_{OP}.Patname=R_{ICU}.Patname) and (Dept= Orthopedics) and (Admit > 1st January) and (Weight > 70).

According to the **Semijoin** strategy the query processing is as follows:

Step-1:
Restrict R_{OP} (Dept= Orthopedics, Admit> 1st January).
Project Patname from restricted $R_{OP.}$
Step-2: Transmit the result R_1 (from Step-1) to ICU site.
Step-3:
Restrict R_{ICU} (Weight > 70) (say N_2 tuples).
Join R_1 and Restricted R_{ICU} (say N_3 tuples).
PROJECT these tuples over the required attributes.
Step-4: Move the Result R_2 (From step-3) to Operation site.
Step-5: JOIN result (R_2) with restricted R_{OP} at Operation site.

Cost analysis of the Semijoin Strategy:
Let the cardinality of the Relation R_1 be N_1 , transmission cost/attribute be T_{cost-a} , cost per tuple comparison be $C_{comp-tuple}$, and cost per tuple concatenation is $C_{cn-tuple}$.

Processing cost at Step-2:
$N_1 * T_{cost-a}$
Processing cost at Step-3:
$N_1 * N_2 * C_{comp-tuple} + N_3 * C_{cn-tuple}$
Processing cost at Step-4:
$N_3 * 3 * T_{cost-a}$ (There are three attributes weight, type_work, and patname)
Processing cost at Step-5:
$N_1 * N_3 * C_{comp-tuple} + N_3 * C_{cn-tuple}$

Hence the total processing cost ($TPC_{semi-join}$) for Semijoin strategy =

$$N_1 * T_{cost-a} + N_1 * N_2 * C_{comp-tuple} + N_3 * C_{cn-tuple} + N_3 * 3 * T_{cost-a} + N_1 * N_3 * C_{comp-tuple} + N_3 * C_{cn-tuple}$$

$$= (N_1 + 3 * N_3) * T_{cost-a} + 2 * N_3 * C_{cn-tuple}$$

$$(N_1 * N_2 + N_1 * N_3) * C_{comp-tuple} \quad (1)$$

According to **JOIN** strategy the same query processing is as follows:
Step-1: Send qualified tuples in Operation (projected over three required attributes) to ICU site.
Step-2: JOIN with the restricted R_{ICU} at ICU site
Step-3: Move the result to Operation site.

Cost analysis of the join Strategy:
Transmission cost at step-1: $3 * N_1 * T_{cost-a}$
Processing cost at step-2: $N_1 * N_2 * C_{comp-tuple} + N_3 * C_{cn-tuple}$
Transmission cost at step-3:
$5 * N_3 * C_{cost-a}$ (as total 5 attributes being transferred)
Therefore total cost is: $3 * N_1 * T_{cost-a} + N_1 * N_2 * C_{comp-tuple} + N_3 * C_{cn-tuple} + 5 * N_3 * T_{cost-a}$
$= (5 * N_3 + 3 * N_1) * T_{cost-a} + N_1 * N_2 * C_{comp-tuple} + N_3 * C_{cn-tuple}$ (2)

So under the assumption that T_{cost-a} (transmission cost/ attribute) contributes much more significantly to the cost than $T_{cn-tuple}$ or $T_{comp-tuple}$. Equation (1) shows that the cost of the semijoin is probably about half of the cost of the join strategy (despite having to JOIN at both sites).

The Structure of FRS [12] **(Fragmentation and Replication strategy):**

The FRS strategy requires one of the relations referenced by a query to be fragmented and other relations to be replicated at the sites that have a fragment of the fragmented relation. 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. Its main feature is to allow parallel processing. Considering Table 1, to process the following query (using FRS strategy) which references fragmented relations R_1 and R_2 and an unfragmented relation R_3:

$Q = \{R_1.A, R_2.C \mid R_1.A = R_2.A \wedge R_2.B = R_3.C\}$.

If site-S_1 is selected for query processing the required steps are:
Step-1: Find the minimum response time of the strategies using only one site (site-1 or site-2 or site-3) to process query.

- Transferring F_{12}, F_{22} to site-1
- Union ($F_{12} \cup F_{11}$) and ($F_{21} \cup F_{22}$)
- JOIN ($F_{12} \cup F_{11}$) $\bowtie$ ($F_{21} \cup F_{22}$) $\bowtie R_3$.

Similarly queries can be processed at site-2, and site-3. The one with minimum response time among these three (sites) is chosen.

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

If R_1 is left fragmented, two sets of sites, {1, 3}, and {2, 3} can be chosen to process the query 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. Therefore R_2 and R_3 are replicated at site-1 and site-3.

Table 1: Placement of Relation in FRS

	Site		
Relation	Site-S_1	Site-S_2	Site-S_3
R_1	F_{11}	F_{11}	F_{12}
R_2	F_{21}	-	F_{22}
R_3	R_3	-	-

Now the query Q is decomposed into two following subqueries:

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

Q_1: $\{F_{12}.A, R_2.C \mid F_{12}.A = R_2.A \wedge R_2.B = R_3.C\}$; where R_2 is the union of F_{21} and F_{22} and the final result of $Q = Q_1 \cup Q_2$. Similarly the response times can be obtained for processing query (Q) at sites 2 and 3.
If R_2 is left fragmented, Q will be processed at sites 1 and 3. Similarly response time can be estimated.

The time spent at site-1 can be shown to be smaller than that of the earlier strategy (step-1) because the fragment F_{12} does not need to be transferred and processed at Site-1. Similarly, the time at site Site-3 can be shown to be smaller than single site processing (Step-1). Thus, the response time if R_1 is left fragmented will be smaller than that in which either all processing takes place in site Site-1 or all processing takes place in site Site-3. Thus the fragment and replicate strategy achieves parallel processing and improves response time. Let m be the total number of fragments, n be the total number of relations. Total number of subqueries (using FRS strategy) at any site is the product of the total number of fragments and replicated relation which is m^{n-1}.

FRS has drawbacks in terms of efficiency and response time. FRS is not applicable to processing distributed queries; in which all of the non fragmented relations are referenced by a query.

Partition and Replicate Strategy (PRS) [15]**:**
Assume the set of sites is S = {S_1, S_2, …..S_m}. The PRS algorithm works as follows. For a given query, the minimum response time is estimated if all referenced data is transferred to and processed at only one of the sites. Next, for each referenced relation and each copy of the relation, the response time is estimated if the copy of the relation is partitioned and distributed to a subset of S and all the other relations are replicated at the sites where they are needed. A choice of processing sites and sizes of fragments for the selected copy of the chosen relation are determined by PRS so as to minimize the response time. Finally, the strategy which gives the minimum response time among all the copies of all the referenced relations is chosen.

Let a query reference two relations R_1 and R_2 which are unfragmented and distributed among three sites S_1, S_2, and S_3 as shown in Table 2. Assume that S_3 is slightly faster than S_2 but much faster than S_1. If the query is processed only at a single site without partitioning any relation, the strategy of sending R_1 from S_1 to S_3 and processing the query at S_3, will give the minimum response time.

Table 2: Distribution of Data for PRS.

Relation	Total tuples$_1$	Site-S_1	Site-S_2	Site-S_3
R_1	8000	R_1	-	
R_2	5000	-	R_2	R_2

Now, if we partition R_1 into two fragments and send the fragment containing 3000 tuples to S_2 and that containing 5000 tuples to S_3, processing at the two sites takes place in parallel. The times incurred at sites S_2 and S_3 are smaller than that of the above strategy since only a fragment is joined with R_2 instead of the whole relation. Therefore, partitioning a relation for parallel processing can reduce response time of a join query.

3. Components and Problems of Distributed Query Optimization

There are three components of distributed query optimization [21]:

Access Method: In most RDBMS products, tables can be accessed in one of two ways: by completely scanning the entire table or by using an index. The best access method to use will always depend upon the circumstances. For example, if 90 percent of the rows in the table are going to be accessed, you would not want to use an index. Scanning all of the rows would actually reduce I/O and overall cost. Whereas, when scanning 10 percent of the total rows, an index will usually provide more efficient access. Of course, some products provide additional access methods, such as hashing. Table scans and indexed access, however, can be found in all of the "Big Six" RDBMS products (i.e., DB2, Sybase, Oracle, Informix, Ingres, and Microsoft).

Join Criteria: If more than one table is accessed, the manner in which they are to be joined together must be determined. Usually the DBMS will provide several different methods of joining tables. For example, DB2 provides three different join methods: merge scan join, nested loop join, and hybrid join. The optimizer must consider factors such as the order in which to join the tables and the number of qualifying rows for each join when calculating an optimal access path. In a distributed environment, which site to begin with in joining the tables is also a consideration.

Transmission Costs: If data from multiple sites must be joined to satisfy a single query, then the cost of transmitting the results from intermediate steps needs to be factored into the equation. At times, it may be more cost effective simply to ship entire tables across the network to enable processing to occur at a single site, thereby reducing overall transmission costs. This component of query optimization is an issue only in a distributed environment.

Major distributed query optimization problems are given below:

Local optimization of semijoins ([16, 22])**:** The issue is to determine the optimal set of semijoins to reduce a single relation. The query optimizer of SDD-1[22] is among the first to apply semijoins to distributed query processing. The optimizer evaluates the benefit and cost of all candidate semijoins, performs the most profitable one, and updates the cardinality of the relation reduced by this semijoin accordingly. The procedure repeats until there is no profitable semijoin available. This approach is greedy and suboptimal. A general version of the above approach is that the query optimizer computes the most profitable set of semijoins to reduce R_i for each relation R_i, instead of the most profitable semijoin. Since all semijoins to R_i , are considered at the same time, local optimality (with respect to R_i) can be attained. The solution obtained is locally optimal in that only the reduction of one relation is taken into consideration at a time.

Join sequence optimization ([18] [23]**):** Using the join sequence method for distributed query processing, some joins are performed locally first, and the resulting relations are then sent to the assembly site. In [23], join sequence optimization is formulated as a graph problem. Suppose there are n relations in the query. Construct a directed graph with $n+1$ nodes, where there is one node corresponding to the assembly site A, and each of the remaining n nodes is one-to-one associated with a relation. An edge (R_i , R_j) means R_i can be sent to R_j to perform a

join. An edge (R_i , A) means R_i can be sent to the assembly site directly. The objective is to find an inversely directed spanning tree toward *A* with the minimal transmission cost. Relations are transmitted to *A* along the path determined by the spanning tree. Each intermediate node on the path represents a join. An example spanning tree is shown in Fig. 2**.** For example, **c**onsider the edge (R_3, R_4), after R_1 and R_2 are joined with R_3, the cardinality of R_3 becomes $(\sigma_1^3 . | R_1 | . | R_3 |) . | R_2 | . \sigma_2^3$; where σ_i^j is the join selectivity which is associated with each join such that $| R_i \bowtie R_j | = \sigma_i^j . | R_i | . | R_j |$. The transmission cost of (R_3, R_4) is therefore $(\sigma_1^3 . | R_1 | . | R_3 |) . | R_2 | . \sigma_2^3 . C_{tp} J_3^4$; where $C_{tp} J_i^j$ is the transmission cost per tuple of sending R_i to R_j . The total cost of the spanning tree in Fig. 2 is thus equal to,

$| R_1 | C_{tp} J_1^3 + | R_2 | C_{tp} J_2^3 +$

$(| R_1 || R_2 || R_3 | . \sigma_1^3 . \sigma_2^3) C_{tp} J_3^4 +$

$(| R_1 || R_2 || R_3 | . \sigma_1^3 . \sigma_2^3 | R_4 | \sigma_3^4) C_{tp} J_4^A$; where

$C_{tp} J_i^A$ is the transmission cost per tuple of sending R_i to A the final assembly (query) site.

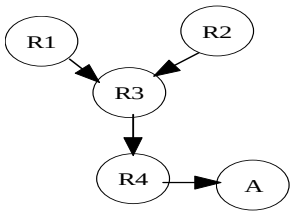

Fig. 2 Spanning Tree for join sequence optimization.

Relation Semijoin on Broadcasting Networks ([24], [25]):

Taking advantage of the broadcasting property of local area networks as shown in [24] and [25], the semijoin method is applied to improve query processing. Using this method, a distributed semijoin is accomplished by sending the entire relation, rather than the join attribute, to its recipient. The advantage is that multiple stations may receive the relation and apply different semijoins at the same time. Further, the assembly site also receives the relation broadcast, and each relation thus needs to be scanned only once. To optimize the transmission cost, the query optimizer has to determine the broadcasting order of the relations.

Semijoin Optimization for Tree Queries ([26], [27]): It has been shown that as far as optimizing the effect of semijoins is concerned, tree queries are fundamentally easier than cyclic queries [16], [28]. Optimization on the semijoin application for tree queries, though less complicated than that for cyclic queries, is still computationally difficult, and precisely NP-hard. **A** relation is said to be fully reduced if, after the execution of a sequence of semijoins, no other semijoins can further reduce its cardinality. A semijoin program is called optimal if it fully reduces the relations required in the join phase with the minimal transmission cost incurred. There are two types of queries, namely single-reducer tree queries and full-reducer tree queries. In single-reducer tree queries, after the semijoin reduction phase, only one relation is needed for final processing, whilst in full-reducer tree queries, after the semijoin reduction phase, all relations are needed for final processing.

4. Proposed Technique for Query Processing in DDBS

Our proposed technique is based on the six definitions D-1 to D-6. We first explain that the straightforward (naive) approach to processing a distributed query would be sending all relations directly to the assembly site, where all joins are performed. This naive method, however, is unfavourable due to its high transmission overhead and low level of parallelism.

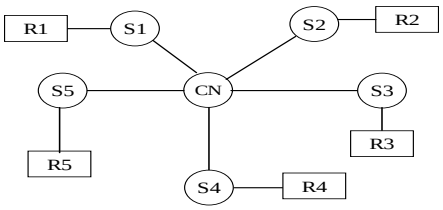

Fig. 3 Relation in different sites.

According to the straightforward approach, the given query (Q): {R_1.A, R_2.B, R_3.C, R_4.D, R_5.E | R_1.A=R_3.A $\wedge$ R_2.B=R_4.B $\wedge$ R_3.C=R_5.C} will be processed at any of the sites shown in Fig.3 and therefore the query is as follows:

Step-1: $[(R_1 \bowtie R_3) \cup (R_2 \bowtie R_4) \cup (R_3 \bowtie R_5)]$

Step-2: Project over the joined relations.

Table 3: Different sites have different relations.

S_1	S_2	S_3	S_4	S_5
F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
F_{21}	F_{22}			
		R_3		
				R_5
			R_4	

Definition-1: Let ***S*** be a set of all the relations referenced by the given query, let ***SPR*** (which is a subset of ***S, SPR*** $\subseteq$ ***S***) be the set of all the distinct relations that exist in the predicate of the referenced query that needs to be fragmented, and let ***SRR*** be the set of replicas that need to transferred, which is the set difference between ***S*** and ***SPR (S \ SPR)***.

Definition-2: Let ***m*** be the number of query processing sites, equal to the total number of the relations ***(N_{TR})*** referenced by the query. The total number of relations ***(TN_{FR})*** needed to be fragmented across ***m*** sites is equal to the number of relations in SPR, and the total number of relations ***(TN_{RP})*** needed to be replicated as to the number of relations in {***S \ SPR***}; where $\{1 \leq TN_{FR} < N_{TR}\}$.

Definition-3: No replica transference is necessary, but the number of relations needed to be fragmented as in definition-2, if $TN_{FR} = N_{TR}$ and the query processing sites ***m*** is equal to the total number of the relations ***(N_{TR})*** referenced by the query.

Definition-4: If the given query has only ***PO*** (project operation) without having any ***WC*** (predicate or where condition), no fragmentation or replication is required. Rather, all the relations referenced by the query are processed at the sites where they are stored. After then all the processed results are combined at any of the ***m*** sites.

Definition-5: If any relation (R) $\subseteq$ ***SPR*** and the relation (R) is already fragmented across ***m*** sites, it is not necessary to fragment that relation (R); whereas if any relation (R) $\subseteq$ ***SPR,*** is fragmented across ***p*** sites which is less than to the number of total ***m*** sites, it is required that relation to be fragmented to the total number of $(m - p)$.

Definition-6:

$$\forall_S : [S_1 ..... S_m] \mid Join(S_{FR} \wedge S_{RP}) \wedge \exists_S \mid \cup_{i=1}^{m} [JR(S_i)];$$

where S_{FR} and S_{RP} represent all the fragmented and replicated relations to any site and JR be the joined relations.

Explanation of the Technique:
Let us consider the query (Q): {R_1.A, R_2.B, R_3.C, R_4.D, R_5.E | R_1.A=R_3.A $\wedge$ R_2.B=R_1.B $\wedge$ R_3.C=R_2.C} and the relations given R_2 in Table 3.

Table 4: Distributed relations using proposed technique.

S_1	S_2	S_3	S_4	S_5
F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
F_{21}	F_{22}	F_{23}	F_{24}	F_{25}
F_{31}	F_{32}	F_{33}	F_{34}	F_{35}
R_4	R_4	R_4	R_4	R_4
R_5	R_5	R_5	R_5	R_5

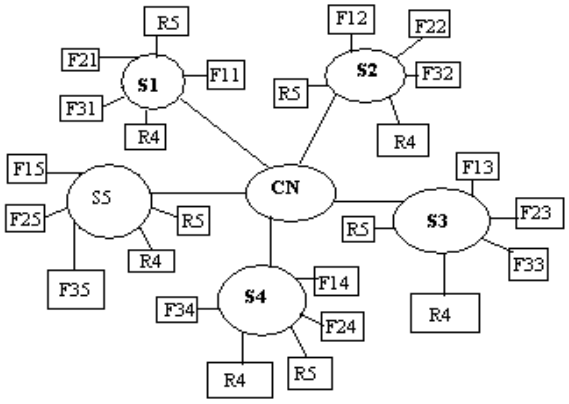

Fig. 4 Fragmented Relations on distributed sites.

According to Definition-1, we have S= {R_1, R_2, R_3, R_4, R_5), ***SPR***= { R_1, R_2, R_3} and ***SRR***={R_4 , R_5 }.
Any given query satisfies either Definition-2 or Definition-3 or Definition-4, and thereafter the query satisfies definition-6.

According to Definition-2, ***m*** = N_{TR} **=5,** TN_{FR} **=3**, TN_{RP} =2, and $\{1 < TN_{FR} < N_{TR}\}$. The given query does not satisfy Definition-3 and Definition-4.

According to Definition-5, relation $R_1 \subseteq$ ***SPR***, and R_1 is already fragmented to all ***m*** sites, hence R_1 won't be required to be fragmented , also relation $R_2 \subseteq$ ***SPR*** , which is fragmented across ***p*** {S_1, S_2} sites that is less than ***m*** , so relation R_2 is required to be fragmented $(m - p)$ sites.

According to the definition, and explanation the distribution of the relations are presented in Table 4, and in Fig. 4. Applying Definition-6, all the fragmented relations and replicated relations are joined to each of the sites and all the joined results are unioned to any of the sites.

4.1 Cost Analysis of the Proposed Query Processing Technique

Let N_1 be the cardinality of each fragmented relation, N_2 be the cardinality of each replicated relation, N_3 be the cardinality of the joined relation, N_{join} be the cardinality of each joined relation , k be the number of attributes in both fragmented and replicated relations, K_{join} be the number of attributes after joining the relations from any site, K_p be the number of attributes to be projected , $T_{cost-att}$ be transmission cost per attribute, TC_{FR} be the fragmented relation transmission cost, TC_{RP} be the replication transmission costs, JC_{ES} be the join cost for each site, CT_{comp} be cost per tuple comparison, CT_{con} be the cost per tuple concatenation, $CP_{per-att}$ be the cost per projected attribute , TT_{JR} be the total transfer cost of joined relation , m be the total number of fragments, and UC be the union cost for all the relations to the destination site.

If the given query satisfies the definition-2, the transmission cost of the fragmented and replicated relations are respectively:

$$TC_{FR} = T_{cost-att} * N_1 * k * m * TN_{FR} \quad (1)$$

$$TC_{RP} = T_{cost-att} * N_2 * k * m * TN_{RP} \quad (2)$$

After satisfying definition-6, the joined cost for each of the site is JC_{ES} =

$$(N_1 * TN_{FR} + N_2 * TN_{RP}) * CT_{comp} + N_3 * CT_{con} \quad (3)$$

The total transferring cost of $(m-1)$ joined relations to the assembly site

$$TT_{JR} = (T_{cost-att} * N_{join} * k_{join}) * (m-1) \quad (4)$$

The union cost

$$UC = \sum_{j=1}^{m} [k_{join} * N_{join} * CT_{con}]_j \quad (5)$$

Therefore the total cost we have (satisfying Definition-2 and Definition-6) using equation (1), (2), (3), (4), and (5).

$$TC_{ARRQ-FP} = TC_{FR} + TC_{RP} + \sum_{j=1}^{m} (JC_{ES})_j + TT_{JR} + UC \quad (6)$$

Similarly if the given query satisfies Definition-3, the transmission cost of the fragmented (no replication transfer necessary) relation is:

$$TC_{FR} = T_{cost-att} * N_1 * k * m * N_{TR} \quad (7)$$

Therefore we have the total cost (satisfying definition-3, definition-6) using equation (7), (3), (4), and (5).

$$TC_{ARRQ-FG} = TC_{FR} + \sum_{j=1}^{m} (JC_{ES})_j + TT_{JR} + UC \quad (8)$$

If the given query satisfies Definition-4, all the relations are transferred to any of the sites, to project and union the relations. Therefore the costs:

$$TC_{PRO-UNI} = \sum_{i=1}^{NT_R} [N_2 * k_p * CP_{per-att}]_i + UC \quad (9)$$

Considering equation (6), (8), and (9) we conclude that $TC_{ARRQ-FG} < TC_{ARRQ-FP} < TC_{PRO-UNI}$, because sending all relations directly to the assembly site, where all joins are performed, is unfavourable due to its high transmission overhead and little parallelism exploitable.

4.2 Cost Analysis of the FRS Strategy

The FRS algorithm [12] requires one of the relations referenced by a query to be fragmented and other relations to be replicated at the sites that have a fragment of the fragmented relation. 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.

Let N_R be the total number of relations, p be the total number of sites, N_1 be the cardinality of the fragmented relation, n be the number of fragments of the fragmented relation , N_2 be the cardinality of each other $(N_R - 1)$ replicated relations, N_3 be the cardinality of the joined relations, k be the number of attributes in both fragmented and replicated relations, n be the number of selected sites to process the query, CT_{comp} be cost per tuple comparison, CT_{con} be the cost per tuple concatenation. The cost for one of the relation to be fragmented across m sites is

$$TC_{O-FR} = N_1 * k * T_{cost-att} * n \quad (10)$$

The cost for all other $(N_R - 1)$ relations to be replicated across n sites is

$$TC_{RP} = (N_2 * k * T_{cost-att}) * (N_R - 1) \quad (11)$$

The local processing costs of these sites are (union cost for fragmented relation and natural join cost)

$$LPC_{AS} = \sum_{j=1}^{p} Cost[\cup_{i=1}^{n} (FR)_i)]_j + [(N_1 * 1 + N_2 * (N_R - 1)) * CT_{comp} + N_3 * CT_{con}] * p \quad (12)$$

Therefore the total cost for FRS strategy is

$$TC_{FRS} = TC_{O-FR} + TC_{RP} + LPC_{AS} \quad (13)$$

4.3 Comparison of Cost Analysis

FRS or PRS requires substantial data transfer and preparation before a query can be processed in parallel at different sites. Performing local reductions before data transfer can reduce the data replication cost and the join processing cost. To achieve this goal, a heuristic Local Reduction (LR) algorithm is presented in [14]. However, local reduction takes time and it delays data transfer. It is not always true that local reduction will reduce the response time for the processing of a given query.

Considering equation (6), (8), and (13) we conclude that $TC_{ARRQ-FG} < TC_{ARRQ-FP} < TC_{FRS}$; because equation (6), and (8) support more than one relation to be fragmented which reduces the size of the relation. This results in lower transmission costs and local processing costs than that supported by equation (13) in which one relation remains to be fragmented.

But comparing equations (9) and (13) it is concluded that $TC_{PRO-UNI} > TC_{FRS}$; as it is always true that sending all relations directly to the assembly site, where all joins are performed, is unfavourable due to its high transmission overhead and little exploitation of parallelism. According to our approach, this case is the worst case depending on the given input query (wherein there are no conditions or predicates in the where clause of the given query).

5 Conclusions and Future Work

The (FRS) fragment and replicate strategy [12] permits parallel processing of a query. Fragments and replicate (FRS) strategies are not applicable for processing distributed queries; in which all the non fragmented relations are referenced by a query. In [29], a new placement dependency algorithm is presented to improve FRS strategy, but the problem of this algorithm, when the number of sites increases is that, 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 the algorithm becomes less favourable. Another drawback of this algorithm is that the average improvement decreases as the number of referenced relations increases though the decrease is rather slow.

In case of FRS strategy, if no relation referenced by a query is fragmented, it is necessary to decide which relation is to be partitioned into fragments; which copy of the relation should be used; how the relation is to be partitioned; and where the fragments are to be sent for processing. To resolve this problem, PRS (Partition and Replicate strategy) is presented in [15]. But PRS algorithm favours joins involving small numbers of relations, and improvement decreases as the number of relations involved in joins increases. Improvement of PRS over single site processing depends heavily on how fast a relation can be partitioned. If the implementation of relation partitioning is inefficient, PRS actually gives worse response time than single site processing.

The technique generally fragments the relations that exist in the predicates (the WHERE condition) of the query. The proposed technique is efficient comparing to other techniques, as more than one relation is allowed to remain fragmented which exploits parallelism, while replicating the other relations (excluding the fragmented relations) to the sites of the fragmented relations. Hence the communication costs and local processing costs are reduced due to the reduced size of the fragmented relations. This also improves the query response time. If the given query references all the relations but only one relation that exists in the predicates (the WHERE condition), this technique works as a FRS strategy which is the worst case situation of the proposed technique.

Experimental results will be reported as the research progresses.

References

[1] P. Valduriez and T. Ozsu, "Principle of Distributed Database Systems.," Prentice Hall, 1999.

[2] C. Liu and C. Yu, "Performance Issues in Distributed Query Processing," *IEEE*, vol. 4:8, pp. 889-905, 1993.

[3] S. Upadhyaya and S. Lata, "Task Allocation in Distributed Computing VS Distributed Database Systems: A Comparative Study," *IJCNS (International Journal of Computer Science and Network Security)*, vol. 8:3, pp. 338-346, 2008.

[4] C. Wang and M.-S. Chen, "On the Complexity of Distributed Query Optimization," *IEEE* vol. 8, pp. 650-662, 1994.

[5] L. M. Hass, "R*:A Research Project on Distributed Relational DBMS," *Database Engineering*, vol. 5:4, 1982.

[6] M. Stonebraker and E. Neuhold, "A Distributed Database Version of INGRES," in *Second Berkley Workshop on Distributed data Management & Computer Networks*, USA, 1977, pp. 19-36.

[7] R. Epstein, M. Stonebraker, and E. Wong, "Distributed Query Processing in Relational Database System," in *ACM SIGMOD* Austin, USA, 1978.

[8] J. S. J. Chen and V. O. K. Li, "Optimizing Joins in Fragmented Database Systems on a Broadcast Local Network," *IEEE Trans. & Software Engineering*, vol. 15:1, pp. 26-38, 1989.

[9] G. Pelagatti and F. A. Schreiber, "A Model of an access Strategy in Distributed Database System," in *International Conference of Database Architecture*, Venice, Italy, 1979.

[10] E. Wong and R. H. Katz, "Distributing a Database for parallelism," in *ACM SIGMOD* San Jose, CA, 1983.
[11] B. Gavish and A. Segev, "Set Query Optimization in Distributed Database System," *ACM TODS*, vol. 11:3, 1986.
[12] C. T. Yu, K. C. Guh, C. C. Chang, C. H. Chen, M. Templeton, and D. Brill, "An Algorithm to Process Queries in Distributed Network," in *IEEE Real-Time Syst. Symp.*, 1984.
[13] C. T. Yu, K. C. Guh, W. Zhang, M. Templeton, D. Brill, and A. L. P. Chen, "Partitioning Relation for Parallel Processing in Fast Local Networks," in *International Conference on Parallel Processing*, 1986, pp. 1021-1028.
[14] C. T. Yu, K. C. Guh, W. Zhang, M. Templeton, D. Brill, and A. L. P. Chen, "An Integrated Algorithm for Distributed Query Processing," in *IFIP Conference on Distributed processing* Amsterdam, 1987.
[15] C. T. Yu, K. C. Guh, W. Zhang, M. Templeton, D. Brill, and A. L. P. Chen, "Partition Strategy for Distributed Query Processing in Local Fast Networks," *IEEE Trans. & Software Engineering*, vol. 15:6, pp. 780-793, 1989.
[16] P. A. Bernstein and W. C. D-M, "Using semi-joins to solve relational queries," *ACM* vol. 28:1, pp. 25-40, 1981.
[17] M. S. Chen and P. S. Yu, "Combining Join and Semi Operations for Distributed Query Processing," *IEEE Trans. of Knowledge & Data Engineering*, vol. 5:3, pp. 534-542, 1993.
[18] M. S. Chen and P. S. Yu, "A Graph Theoritical Approach to Determine a Join Reducer Sequences in Distributed Query Processing," *IEEE Trans. of Knowledge & Data Engineering*, vol. 6:1, pp. 152-165, 1994.
[19] M. S. Chen and P. S. Yu, "Interleaving a Join Sequence with Semijoins in Distributed Query Processing," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3:5, pp. 611-621, 1992.
[20] T. Ibaraki and T. Kameda, "On the Optimal Nesting Order for N-Relatioanl Joins," *ACM (TODS)*, vol. 9:3, pp. 482-502, 1984.
[21] C. S. Mullins, "Distributed Query Optimization," 1996.
[22] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems, ,* vol. 6: 4, pp. 602-625, 1981.
[23] K. T. Huang, "Query Optimization in Distributed Databases," in *Laboratory for Information and Decision Systems*: MIT, 1982.
[24] A. R. Hevner, O. Q. Wu, and S. B. Yao, "Query Optimization on Local Area Networks,," *ACM Trans. Ofice Information*, vol. 3, pp. 35-62, 1985.
[25] W. Perrizo, J. Y. Li, and W. Hoffman, " Algorithms for Distributed Query Processing in Broadcasting Local Area Networks," *IEEE Trans. Knowledge and Data Eng.*, vol. 1:2 pp. 215-225, 1989.
[26] A. L. P. Chen and V. O. K. Li, "Improvement Algorithms for Semijoin Query Processing Programs in Distributed Database Systems," *IEEE Trans. Comput. ,* vol. 33:11, pp. 959-967, 1984.
[27] D. M. Chiu and Y. C. Ho, "A Methodology for Interpreting Tree Queries into Optimal Semi-loin Expressions " in *Proc. ACM SIG-MOD*, 1980, pp. 169-178.
[28] N. Goodman and Shmueli, "Tree Queries: A Simple Class of Relational Queries," *ACM Trans. Database Systems*, vol. 7:4, pp. 653-677, 1982.
[29] C. Liu, H. Chen, and W. Krueger, "A Distributed Query Processing Strategy Using Placement Dependency," in *Internation Conference on Data Engineering*, LA, USA, 1996, pp. 477-484.

B.M. Monjurul ALom who born in Bagherpara, Jessore, Bangladesh, is a research (PhD) student in the School of Electrical Engineering and Computer Science, The University of Newcastle, Australia. Mr Alom has completed his MSc engineering degree from Bangladesh University of Engineering and Technology, Dhaka. His research interest is Distributed (Structured and Semistructured) Database Management. Mr. Alom was an assistant professor in CSE dept from 2004 to 2007 and a lecturer from 2000 to 2004 in Dhaka University of Engineering and Technology, Gazipur, Bangladesh.

Dr. Frans Henskens is an Associate Professor in the School of Electrical Engineering and Computer Science, Newcastle University Australia. He is also Head, Discipline of Computer Science & Software Engineering, Deputy Head, School of Electrical Engineering & Computer Science, and Assistant Dean IT in Faculty of Engineering & Built Environment. His research interests include engineering of flexible software systems, bioinformatics, operating systems and computer forensics, distributed and grid computing, resilience and availability in database systems.

Dr. Michael Hannaford is Assistant Dean (Postgraduate Studies) of FEBE, and a Senior Lecturer in the School of Electrical Engineering and Computer Science at the University of Newcastle. His research interests are in the areas of Distributed Computing, and Programming Language Design and Implementation.