International Journal of Database Theory and Application
Vol.10, No.6 (2017), pp.13-34
http://dx.doi.org/10.14257/ijdta.2017.10.6.02

Integrated Design Solution for Distributed Databases Using Genetic Algorithms

Sukkyu Song

Youngsan University
sksong@ysu.ac.kr

Abstract

The design of distributed database systems has prompted many research problems. Among others, the issue of interdependency and interaction associated with data fragmentation, data allocation, and distributed query optimization still remains unanswered. These problems have been proven to be NP-complete or NP-hard, so most previous studies have addressed these problems in isolation by making simplified assumptions. However, these problems are interdependent and hence solving them independently results in inefficient solution overall. In this research, we develop an integrated distributed database design solution for three problems: partitioning data sets, allocating partitioned data sets among the sites of a network, and allocating operations as a problem of distributed query optimization. We use a transaction-based approach, wherein most important transactions are considered in determining the effective design of distributed database, and consider two types of transactions: OLTP (on-line transaction processing) and DSS (decision support system), for reflecting various distributed database design objectives such as total time minimization, response time minimization, and minimization of a combination of both. We employ genetic algorithms as searching methods for the best distributed database design solution. The integrated design solutions are determined by analyzing interactions between the problems in four stages: 1) between vertical fragmentation and operation allocation, 2) between vertical fragmentation and data allocation, 3) between data allocation and operation allocation, and 4) integration of all three problems, with the objectives of cost minimization and load balancing. Our integrated approach resulted in a cost effective distributed database design compared to the designs considering the problems in isolation.

***Keywords**: Distributed database, Data fragmentation, Operation (subquery) allocation, Data allocation, Total time, Response time, Load balancing, Genetic algorithm*

1. Introduction

The design of a distributed database system has presented to researchers and system designers many challenges, some of which are related to managerial design problems, including decisions on data fragmentation and data allocation, and some of which are related to technical design problems, including query optimization, concurrency control, failure recovery, and integrity and security [3-4, 8-9, 15]. Many models and solution techniques have been proposed to solve these managerial and technical problems. In most previous studies, however, various aspects of these problems have been evaluated in isolation even though these problems are interdependent. Among various interdependencies associated with components of a distributed database system, the data distribution problem, concerning data fragmentation, data allocation, and operation allocation, is the one being typically evaluated separately even though the interdependencies among these problems are well recognized. In fact, the individual problems had been shown to be NP-hard or NP-complete [6]. Thus, in an effort to reduce the computational complexity and ensure tractability, these individual problems are treated in isolation by making simplified assumptions on overall distributed database system functions.

In this research, we propose a new integrated solution approach to the design of distributed database system that integrates the following three design decisions: data fragmentation, data allocation, and operation allocation in coordination with data replication while these design decisions are satisfying the physical and managerial requirements [15]. The physical requirements might be the capacity constraints of system resources such as CPUs, I/Os, and communication channels. The managerial requirements might be data availability, system reliability, average transaction response time, and systems security. The problems mentioned are classified into combinatorial optimization problems, and these problems are shown to be NP-hard or NP-complete. In order to solve these problems, we use a genetic algorithm as an alternative heuristic approach [1], [2], [10], [11], and [12].

This paper is organized as follows. In Section 2, we present the integrated design method for distributed databases. Section 3 has discussion of cost models for genetic algorithms, including vertical fragmentation, query and update processing model and analytical cost models for total time, response time and load balancing. Section 4 has illustration of our integrated solution method, and the result obtained by genetic algorithms. Section 5 has conclusions.

2. Integrated Design Method

Figure 1 show an integrated design method used in this research. The details of each step are presented in the sections below.

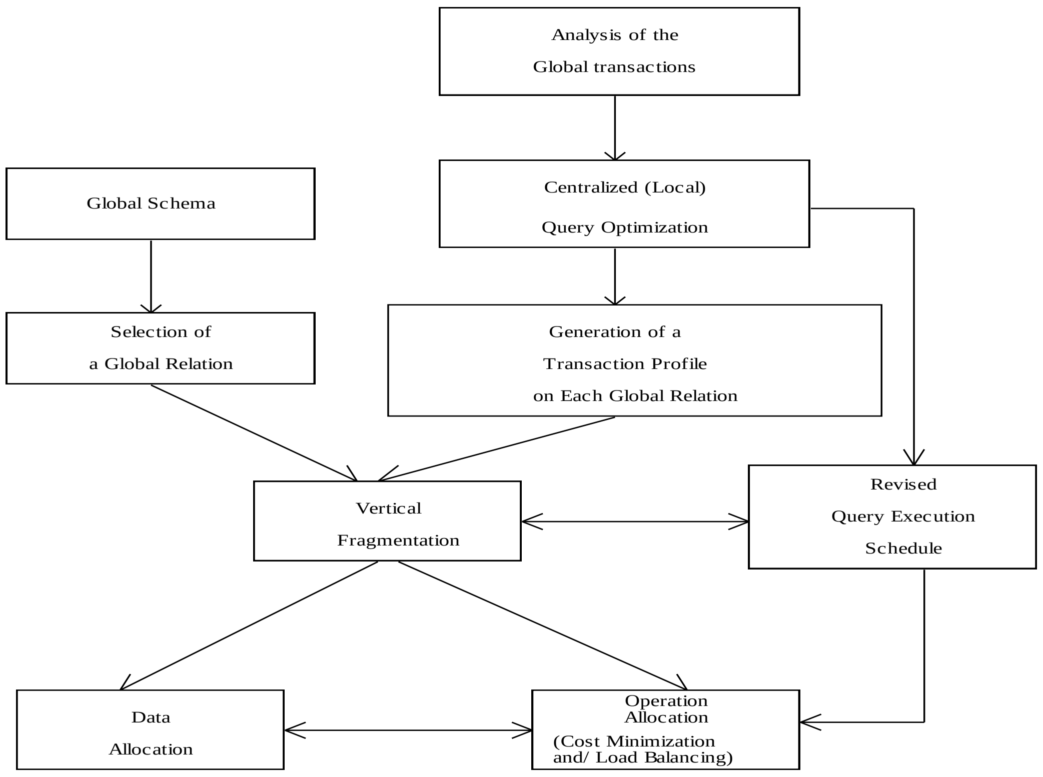
Analysis of the Global transactions
Global Schema
Centralized (Local) Query Optimization
Selection of a Global Relation
Generation of a Transaction Profile on Each Global Relation
Vertical Fragmentation
Revised Query Execution Schedule
Data Allocation
Operation Allocation (Cost Minimization and/ Load Balancing)

Figure 1. Integrated Design Method

2.1. Vertical Fragmentation

The rationale of vertical fragmentation is to combine attributes frequently used together. One group of attributes may be mainly used in one local site, and another group in another local site. Obviously, partitioning the relation into fragments and locating one fragment at the local site and the other at the other site will tremendously decrease network traffic.

As shown in Figure 1, a global schema is assumed to be developed already, and hence we have a set of relations. The global schema only describes the logical relationships between data and do not reflect the way the data are processed. To determine which attributes of a relation should be grouped together, we have to analyze all transactions in which this relation is used. Two approaches are commonly used to determine which transactions to consider in the design process: either all transactions or a dominant subset [2]. In this research, we will take the dominant subset approach. A dominant subset is commonly selected on such criteria as high frequency of execution, high volume of data accessed, response time constraints, and explicit priority. Once the dominant transactions are selected, we need to generate a transaction profile. A transaction profile is a matrix which has attributes as columns and transactions as rows with the access frequency of the transactions. The transaction profile for each relation considered in turn can be generated based on the query execution schedule. In order to find a query execution schedule for each query transaction, we use a centralized query optimization technique as if all relations are placed in one local site since at this stage of design process, we do not know how the fragments are allocated to sites. Transaction profiles generated in this step will be used as inputs to vertical fragmentation design.

In our vertical fragmentation algorithm, the local transaction processing cost will be determined by selecting the minimum cost access path for a given particular partitioning scheme. After relations are partitioned into fragments, these fragments become the unit of allocation for the next step.

2.2. Data Allocation

Once the unit of allocation is determined at the previous step, the allocation of fragments to local sites is the next design process. The "goodness" of a particular allocation scheme will be measured based on the total operating cost of all dominant transactions (see the section below). In order to determine each query transaction cost, the query execution schedule will be revised based on the partitioning scheme for each relation determined in the previous step. The revised query execution schedule will be used as an input to the operation allocation step. In other words, for each query transaction, the processing cost can be calculated based on the allocation scheme selected, which in turn is based on the query execution schedule. The sum of all transaction processing costs is the measure of the "goodness" of a particular data allocation.

As the genetic algorithm generates many possible data allocation schemes, the total transaction processing cost will be calculated for each allocation scheme. Each allocation scheme will then be compared with each other, and the optimal data allocation scheme will be selected accordingly.

2.3. Operation Allocation

2.3.1. Cost Minimization

As mentioned in the above section, the revised query execution schedule and data allocation will be fixed in order to find an operation allocation scheme. In order to evaluate the "goodness" of a particular operation allocation scheme, the total operation (subquery) execution cost should be measured. The operation execution cost will be calculated in terms of time units, which refer to using resources such as CPUs, I/Os and communication channels. In this research, operation execution cost is measured with respect to either total execution time (total time) or response time based on the characteristics of each transaction. OLTP (on-line transaction processing) types of transaction will be measured by the total time whereas DSSs (decision support systems) type transactions will be measured by response time. The total time is calculated by summing all operation execution times, *i.e.*, both local processing times and data transmission times. The response time is the elapsed time from the initiation to the completion of the query, including transmitting the query results back to the site where the query has originated.

The total cost of all transaction executions in terms of total time, response time, or the combination of both is actually the measure of the "goodness" for a particular data allocation scheme.

2.3.2. Load Balancing

In this research, we do not explicitly consider load balancing as another objective of operation allocation. We instead consider total cost minimization as the primary objective and load balancing as the secondary objective. In order to accomplish this, we first solve the operation allocation problem whose objective is total cost minimization, then the CPU and I/O loads at each site are calculated according to operation allocation schemes; that is, operation allocation for load balancing is not determined by any measure of load balancing itself, but by cost minimization (note that in order to measure the degree of load balancing among network sites, we define the unbalanced factor as the sum of the absolute deviation of sites loads from the average network load).

Like operation allocation mentioned in the previous section, the interaction between load balancing and data allocation will be also considered in this research [5], [7], and [14].

3. Development of Cost Models for Genetic Algorithms

3.1. Cost Model for Vertical Fragmentation

This section presents the cost model to the vertical fragmentation problem in conjunction with access path selection to be used for a binary vertical fragmentation genetic algorithm. We assume that the vertical fragmentation is designed for each single relation independently from other relations in the database, so the fragmentation is strict in the sense that attributes in the partitioned fragments are not allowed to overlap. At the first iteration of the genetic algorithm, two distinct (binary) fragments are produced, and then the same algorithm is applied recursively to further partition the fragment produced at the previous iteration until no improved fragmentation is possible.

The objective of vertical fragmentation is to minimize the total transaction execution cost (total cost) by partitioning a relation into two or more fragments. We define the total cost of a binary fragmentation scheme as the sum of disk accesses incurred by each transaction. To calculate the total cost for the given binary fragmentation scheme, the number of disk accesses incurred by each transaction should be determined first. And to determine the number of disk accesses of each transaction, the access path for each transaction should be identified for the given binary fragmentation scheme. The access path selection is done by evaluating the costs (the number of disk accesses) estimated for all available access paths and selecting the one with minimum cost. In this paper, we consider three representative access paths, *i.e.*, sequential scan, clustered index scan, and unclustered index scan, as used in [2], [8], [11], and [13].

Attributes of a relation are classified into three categories: clustering (primary key), unclustering (secondary key), and non-key. We assume that a relation (hence, fragments) is stored as an ordered set of contiguous tuples (records) based on the primary key. It is also assumed that the selection of the unclustering attributes to be inverted is determined exogenously prior to the determination of vertical fragmentation. The restrict attribute is the one being appeared in the selection formula (or predicate) of an SQL statement. The scan attribute is the one being used to actually scan the relation. If there is one restrict attribute, it is the same as the scan attribute, but if there are multiple restrict attributes, then the scan attribute is selected from those restrict attributes (see the next section for details as to how to select the scan attribute). The selectivity of each restrict (or scan) attribute is defined as the ratio of the number of tuples satisfying its selection predicate and the total number of tuples of the relation. We assume that the selection formula consists of only conjunctive predicates since any disjunctive predicates can be converted into conjunctive ones before processing a transaction.

In summary, it is assumed that a selected set of important transactions against a relation is defined a priori, and that the following information about a transaction is available:

(1) The frequency of the transaction per unit time,

(2) The subset of the attributes accessed by the transaction, and

(3) The selectivity of each restrict attribute.

After a relation is partitioned, each fragment will have a record for each tuple of the original relation, and tuples are assumed to be of fixed length. The tuples in different fragments can be identified by replicating the primary key in all fragments or by using tuple identifiers (TIDs), *i.e.*, system controlled identifiers for each tuple of the original relation, which are replicated into every fragment. In this paper, we assume that a tuple is identified by a tuple identifier which has two components: page number and offset so that any tuple can be accessed directly based on TID.

The number of disk accesses of a transaction is equal to the number of disk accesses per run multiplied by its execution frequency. As mentioned, the number of disk accesses per run of a transaction depends on the access path it uses to scan the relation. For an index scan, the average number of disk accesses incurred by a transaction depends on the average fraction of tuples that satisfied the predicate of the indexed attribute. If the indexed attribute is the clustering attribute, it is called a cluster index scan. Otherwise, it is an unclustered index scan. A sequential scan retrieves all pages of a relation. Since retrieving is sequential, several pages of the relation can be prefetched by a single disk access.

Once the relation is partitioned into two fragments, the fragment containing the scan attribute of a transaction is referred to as the primary fragment for that transaction, otherwise it is called the secondary fragment. For a given transaction, the primary fragment being scanned may or may not contain all of the required attributes. The number of disk accesses is therefore expressed as a sum of two components:

(1) The number of disk accesses required to retrieve tuples from the primary fragment, and

(2) The number of disk accesses required to retrieve the remainder of the original tuples from the secondary fragment

Primary Fragment Access Cost

Let T_1 be the number of disk accesses required to scan the primary fragment and attribute A_S be the scan attribute of a transaction, and so it will be used as the basis of scanning the relation.

For a clustered index scan, a clustering attribute (A_1) is used as the scan attribute (A_S), and we assume that the clustering index page is resided in main memory all the time. Then the number of disk accesses can be estimated as

$$T_{1C} = S_{A1}\ C_R\ L^P / P$$

where

S_{A1} is the selectivity of the selection predicate on the scan attribute A_1,

C_R is the cardinality of the relation *R*,

P is the page block size (4k bytes), and

L^P is the tuple size of the primary fragment in bytes, including the tuple identifier (L_{ID}).

For an unclustered index,

$T_{1U} = M(1 - (1 - 1/M)^K) + S_{A_S} N_I$

$M = C_R L^P / P$

$K = S_{A_S} C_R$

where

M is the page size of the primary fragment, including the tuple identifier (L_{ID}),

K is the number of tuples satisfying the selection predicate on the scan attribute A_S,

S_{A_S} is the selectivity of the selection predicate on the scan attribute A_S, and

N_I is the number of pages in the scan index.

Note that for the unclustered index scan, we adopt the formula proposed in [2], which is a good approximation when $K << C_R$ and $M << C_R$, but if K is large, the sequential scan will be selected instead of the unclustered index scan. When there are more than one restrict attributes in the primary fragment, the scan attribute (A_S) for the unclustered index scan is the one that has the minimum selectivity among those restrict attributes indexed already other than the clustering attribute. Thus, the selectivity (S_{A_S}) in above equations is that of the scan attribute selected. Since we assume that the selection formula consists of conjunctive predicates, all tuples satisfying the selection predicate can be identified by scanning a fragment using the indexed attribute having the minimum selectivity. Note, however, that the actual number of tuples satisfying the conjunctive predicate will be based on the overall selectivity of multiple restrict attributes.

For a sequential scan,

$T_{1S} = C_R L^P / (P B)$

where B is the prefetch blocking factor

In sum, if the scan attribute of the primary fragment is the clustering attribute, then

$$T_1 = Min\{T_{1C}, T_{1S}\}$$

else if the scan attribute is the unclustering attribute, but the clustering attribute is one of the restrict attributes, then

$$T_1 = Min\{T_{1C}, T_{1U}, T_{1S}\}$$

else if the scan attribute is the unclustering attribute, but the clustering attribute is not the restrict attribute, then

$$T_1 = Min\{T_{1U}, T_{1S}\}$$

<u>Secondary Fragment Access Cost</u>

Let T_2 be the number of disk accesses required to scan the secondary fragment. And assume that there are *g* restrict attributes, *i. e.*, attributes appeared in the selection formula. Let A_1, A_2,..., A_g be the *g* restrict attributes. Define S_{Ai} be the selectivity of the selection predicate on attributes A_i. Assuming the attribute values are uncorrelated or independent, the overall selectivity of the multiple restrict attributes is the product of the individual selectivity of each restrict attribute. Then the number of tuples selected from the primary fragment can be expressed as

$$S^P = S_{A_S} \prod_{i=2}^{g} S_{Ai}$$

The product term, S_{Ai}, takes the value S_{Ai} if A_i is in the same (primary) fragment as A_S (scan attribute). Otherwise, it is equal to 1.

If attributes not in the primary fragment are required, additional disk accesses are needed to retrieve tuples in the secondary fragment. We can consider three cases depending on what kind of access path is used to scan the primary fragment:

For a sequential scan, the secondary fragment can be accessed either by a sequential scan or through tuple identifier. If S^P is small, the tuple identifiers can be used to retrieve the remaining tuples in the secondary fragment intuitively. Otherwise, a sequential scan can be used instead. Using a tuple identifier on the secondary fragment, the number of disk accesses can be estimated as

$$T_{2I} = M(1 - (1 - 1/M)^K)$$

$$M = C_R L^S/P$$

$$K = S^P C_R$$

where

L^S is the tuple size of the secondary fragment in bytes, including the tuple identifier.

If a sequential scan is used for the secondary fragment, then

$$T_{2S} = C_R L^S/(P\,B)$$

The number of disk accesses for the secondary fragment can then be expressed as

$$T_2 = Min\,\{T_{2I}, T_{2S}\}$$

If an unclustered index scan is used to access the primary fragment, the secondary fragment can be accessed using either tuple identifiers or sequential scan, depending upon which incurs fewer disk accesses so that

$$T_2 = Min\,\{T_{2I}, T_{2S}\}$$

If a clustered index scan is used for the primary fragment, the secondary fragment can be either accessed through tuple identifiers identified using the clustered index in the primary fragment or sequential scan, depending upon which incurs fewer disk accesses. If tuple identifiers are used to access the secondary fragment, it would be similar to a clustered index scan on the primary fragment, so that

$$T_{2C} = S^P C_R L^S/P$$

Thus for case of a clustered index scan on the primary fragment, the disk accesses to the secondary fragment can be estimated as

$$T_2 = Min\,\{T_{2C}, T_{2S}\}$$

Note that when there are multiple restricted attributes in the selection formula, it is possible that these attributes are split into both fragments. If it is happened, it is possible to process two fragments in two different ways depending on which fragment is accessed first (or which one becomes the primary fragment), and as a result, the number of disk accesses will be also different. Thus, in this case, the number of disk accesses required by two access methods is compared, then choose the minimum one as the access method to process two fragments.

3.2. Cost Models for Operation Allocation

3.2.1. Total Time Model

The total time for each query is the sum of local processing times and communication times for all subqueries. Total Time = $\sum_j (\mathrm{LP}_j^k + \mathrm{COM}_j^k)$, where LP_j^k represent the local processing time of the subquery j (a node in the query tree in Figure 2) of a query k. COM_j^k represents the communication time of transmitting the input relation(s) to the site at which the subquery j of a query k is being executed.

3.2.1.1. Local processing time (LP_j^k)

The local processing time of a subquery depends on an operation type, the size of the input relation(s), the CPU speed and the I/O speed of the site selected. We assume that CPU processing is proportional to the amount of data accessed and that I/O time is proportional to the number of blocks read or written.
(A) For a selection or projection on a relation, the local processing time for the subquery j of the query k is defined as:

$$LP_j^k = \sum_t Y_{jt}^k \, (IO_t \sum_i Z_{ij}^k B_{ij}^k + CPU_t \sum_i Z_{ij}^k B_{ij}^k) \tag{1}$$

where B_{ij}^k is the number of blocks of relation i accessed by subquery j of query k,
IO_t is the I/O time of site t in msec for transferring 4k byte page into main memory,
CPU_t is the CPU time of site t in msec per 4k byte page for selection and/or projection.

(B) We also assume that the intermediate result of each unary or join operation is transmitted directly to the next join site and stored at the next join site before the execution of the next join operation. As such, the local processing time for the join j of the query k is defined as:

$$LP_j^k = \sum_t Y_{jt}^k \, IO_t \sum_m \sum_i \rho_m \, Z_{ijp[m]}^k B_{ijp[m]}^k + \tag{2a}$$

$$\sum_t Y_{jt}^k \, (IO_t \prod_i Z_{ij}^k B_{ij}^k + CPU_t \prod_i Z_{ij}^k B_{ij}^k) \tag{2b}$$

where ρ_m represents the selectivity of the two previous operations (m = 1 or 2),
where the selectivity is the ratio of output relation size and input relation size, and
$B_{ijp[m]}^k$ is the size of an input (intermediate) relation where p[m] represents two previous operations of the join operation j (m is 1 for the left and 2 for the right operation).

Note that ρ_m can represent selection, projection or join selectivity. (2a) represents the I/O time to store the intermediate results of the previous operations to the site of the current join operation. (2b) represents the I/O and CPU processing times for the current join operation. Note that we convert $B_{ijp[m]}^k$ (the size of intermediate results being stored at the join site) to B_{ij}^k (the size of same intermediate results being retrieved for the current join operation) for notational convenience so that B_{ij}^k will be used for the next join operation with the join selectivity of the current join operation.

3.2.1.2. Communication Time (COM_j^k)

When either of the relation(s) to be joined is not produced at the site at which the join operation is performed, communication for join operations is needed, and is expressed as follows:

$$COM_j^k = \sum_m \sum_t \sum_p Y_{jp[m]t}^k \, Y_{jp}^k \, C_{tp} \, (\sum_i Z_{ijp[m]}^k B_{ijp[m]}^k)$$

where C_{tp} is the communication cost between site p and site t in msec per 4k byte page.

Note that if a previous operation and the join operation are executed at the same site (t=p), then C_{tp} =0. Communication for sending the final result is also needed if the final operation is not performed at the query originating site. Since there is only one previous operation for the final operation, we assume that $Z^{k}_{ijp[2]}$ for all i is 0 (also $B^{k}_{ijp[2]}$ = 0). It should be noted that we consider communication cost to include data transmission cost. However, in real world, communication cost may also include time to synchronize the two CPUs. In this research, we ignore this synchronization time, since this is usually a fixed overhead cost and it is not variable like data transfer cost.

3.2.2. Response Time Model

In a partially replicated distributed database system, it is possible to decompose a query into subqueries that can be processed in parallel and also their intermediate relations can be transmitted in parallel to the required site. Two types of parallel execution are possible: (1) intra-operation parallelism, and (2) inter-operation parallelism [6]. A typical example of intra-operation parallelism is pipelining of a single join operation, by which two sites work in parallel; that is, the site that request remote data will begin its join processing as soon as the first tuple or packet of data has arrived, whereas in sequential processing, the site receiving data will not begin its join processing until all of the required data has arrived. Inter-operation parallelism refers that several subqueries in a single query can be executed in parallel. In this research we assume the join operation is performed using the sequential processing method, and we are concerned only with parallelism in a single query, not among multiple queries.

Response time is calculated by taking into consideration the possibility of performing local processing and data transmission in parallel under the condition that the operations are performed at different sites as mentioned in the previous section. The response time of query k is:

Response time RT^{k}_{j} = $COM^{k}_{j}(p[1])$ + LP^{k}_{j} (p[1]) + RT^{k}_{j} (p[1])

where RT^{k}_{j} (p[1]) is the recursive function for the response time.

The first term $COM^{k}_{j}(p[1])$ is to calculate the communication time sending the results to the query originating site ($Z^{k}_{ijp[2]}$ for all i is 0 and $B^{k}_{ijp[2]}$ = 0) and the LP^{k}_{j} (p[1]) refers to the local processing time of the final operation. For the recursive function RT^{k}_{j} (p[1]) (but we will use RT^{k}_{j} for convenience), we calculate the cost as follows. Four scenarios exist depending upon sites at which the join operation j and the two preceding operations p[1] and p[2] are executed.

3.2.2.1. Scenario – 1:

The join operation j and the sites two preceding operators p[1] and p[2] are executed at the same site; that is, $Y^{k}_{jp[1]t}Y^{k}_{jp[2]t}C_{tp}=0$, $Y^{k}_{jp[1]t}Y^{k}_{jt}C_{tp}=0$ and $Y^{k}_{jt}Y^{k}_{jp[2]t}C_{tp}=0$ then RT^{k}_{j} can be calculated by using the equation.

LP^{k}_{j} + $\sum_{m} LP^{k}_{j}(p[m]$ + $RT^{k}_{j}(p[m])$)

Here, LP^{k}_{j} is the local processing time for sub query j, $LP^{k}_{j}(p[m])$ is the local processing time for the preceding left (m=1) or right (m=2) operation (*i.e.* subsub query). These local processing times are calculated using the equations introduced in the previous section. $RT^{k}_{j}(p[m])$ is the (response) time when a preceding operator is available for local processing.

3.2.2.2. Scenario – 2:

The join operation j and the two preceding operators p[1] and p[2] are performed at three different sites. In this case the three operators can be run in parallel. Then the response time of the entire group is computed as the maximum of resource consumption of individual operators and the usage of all the shared resources (such as communication times) [6]. Then RT_j^k is given by

$$\text{Max } \{ LP_j^k, \tag{3a}$$

$$LP_j^k(p[1]) + RT_j^k(p[1]), \tag{3b}$$

$$LP_j^k(p[2]) + RT_j^k(p[2]), \tag{3c}$$

$$COM_j^k(p[1]) + COM_j^k(p[2]) \tag{3d}$$

where $COM_j^k(p[1]) = Y_{jp[1]t}^k Y_{jp}^k C_{tp} (\sum_i Z_{ijp[1]}^k B_{ijp[1]}^k)$

$COM_j^k(p[2]) = Y_{jp[2]t}^k Y_{jp}^k C_{tp} (\sum_i Z_{ijp[2]}^k B_{ijp[2]}^k)$

In the above, (3d) represents shared resource consumption, which is the communication time. (3a) is the local processing time for subquery j and (3b) and (3c) are the processing times for the two preceding operations of subquery j. The communication costs will be additive, since those are the overheads on the receiving node, as represented by (3d).

3.2.2.3. Scenario – 3:

The sites at which two preceding operations of subquery j are performed are different and the join subquery j uses one of these sites. There is no communication cost between one of the preceding operators, say p[1], and the operator j. That is, $Y_{jp[1]t}^k Y_{jt}^k C_{tt} = 0$, $Y_{jp[2]p}^k Y_{jt}^k C_{tp} \neq 0$ and $Y_{jp[1]t}^k Y_{jp[2]p}^k C_{tp} \neq 0$, then RT_j^k is given by:

$$\text{Max } \{ LP_j^k + LP_j^k(p[1]) + RT_j^k(p[1]), \tag{4a}$$

$$LP_j^k(p[2]) + RT_j^k(p[2]), \tag{4b}$$

$$COM_j^k(p[2]) \} \tag{4c}$$

where $COM_j^k(p[2]) = Y_{jp[2]p}^k Y_{jt}^k C_{tp} (\sum_i Z_{ijp[2]}^k B_{ijp[2]}^k)$

In the above since sub query j and the left previous operation p[1] are executed at the same site, the local processing times of the two sites need to be added (4a). Since right previous operation p[2] is executed at a different site, its local processing time (included in (4b)) can be executed in parallel. In addition, the communication time (4c) can be implemented in parallel as well.

3.2.2.4. Scenario – 4:

In secenario-4, the two preceding operations of subquery j, p[1] and p[2], are executed at the same site, while the subquery j is executed at a different site. There is communication time involved in shipping data from both the preceding operations p[1] and p[2] to the site of subquery j. That is, $Y^k_{jp[1]p}Y^k_{jt}C_{tp} \neq 0$, $Y^k_{jp[2]p}Y^k_{jt}C_{tp} \neq 0$ and $Y^k_{jp[1]p}Y^k_{jp[2]p}C_{pp} = 0$. Also, there will be no parallelism between the operations p[1] and p[2]. Then RT^k_j is given by

$$\text{Max } \{ LP^k_j , \tag{5a}$$

$$LP^k_j(p[1]) + LP^k_j(p[2]) + RT^k_j(p[1]) + RT^k_j(p[2]), \tag{5b}$$

$$COM^k_j(p[2]) + COM^k_j(p[2]) \} \tag{5c}$$

where $COM^k_j(p[1]) = Y^k_{jp[1]p}Y^k_{jt}C_{tp} (\sum_i Z^k_{ijp[1]}B^k_{ijp[1]})$

$$COM^k_j(p[2]) = Y^k_{jp[2]p}Y^k_{jt}C_{tp} (\sum_i Z^k_{ijp[2]}B^k_{ijp[2]})$$

In the above, since subquery j is executed at a different site than the preceding operators, its local processing of subquery j (5a) can be done in parallel to the communication time (5c) and the processing times of p[1] and p[2] . Since the preceding operators are executed at the same site, their local processing times are additive (5b). Also, the communication costs will be additive, since those are the overheads on the receiving node. Above equations hold whether previous operations are joins, selections, or projections, or other relational algebra operators.

The stopping condition of the recursive function RT is as follows. We define: if p[m] in $Z^k_{ijp[m]}$ is equal to zero in the response time recursive function, where zero for p[m] means that the previous operation for this operation j (subquery) is original relation. In scenarios 2 and 3, parallelism between the preceding operations p[1] and p[2] is implied. It is assumed there is no clash in data access between the two preceding operations, *i.e.*, $Z^k_{ij}(p[1]) * Z^k_{ij}(p[2]) = 0 \ \forall_i$, otherwise local processing times can be additive in the worst case.

3.2.3. Query Tree and Update Tree Model for Update Transaction

A query tree is illustrated in the query part in Figure 3. A node is called a leaf node (F1 and F2) if it has no incoming arcs; that is, it represents the relations in the database. A node is called an operation node (nodes 1, 2 and 3) if it has incoming and outgoing arcs. The operation nodes represent the relational operations. The operation nodes such as 1 and 2 represent a unary operation such as selection, projection or a combination of both, and the operation node such as 3 represents a binary operation such as join or union. Sometimes a binary operation is performed on an input relation directly without any unary operation(s), and in this case the unary operation node connected to the corresponding input relation is called a dummy operation node. An operation node without any outgoing arcs is called a result node (node 4). An arc represents the transmission of a (intermediate) relation into the operations, such as f3, f4 and f5.

There is a site set associated with each node in the query tree. The members of the site set for a leaf node are those sites that hold a copy of that relation. The site set for an operation node contains those sites that can perform the operation. In general, selection and projection operations requiring relations should be executed at only those sites that hold a copy of relations referenced so that there is no transmission of a relation required at the site of the operations, but join operations can be executed at any site.

An update transaction may be viewed as a two-part action, wherein the first part corresponds to a query transaction, followed by the second part which updates the value of a set of relations, as shown in Figure 3. The simplified SQL statement for the update query tree Figure 3 may be as follows:

```
UPDATE F3, F4 Alias F
SET         F.z = F.z * 1.1
WHERE  F.k IN (SELECT   k
                      FROM     F1, F2
                      WHERE  F1.x = F2.y)
```

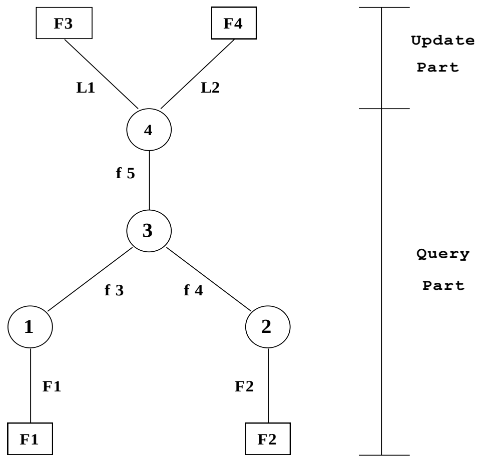

Figure 3. Query Tree for Update Transaction

In the second part of an update transaction, the update values (L1 and L2, which are the same as the intermediate relation f5 resulting from the final operation 3 in Figure 3) resulting from the first part must be sent from the update initiation site (site for operation 4 in Figure 3) to all sites that have a copy of the relation being updated, and then the relation must be updated at each site (for example, two copies of F3 and three copies of F4 in Figure 3), which incurs CPU and I/O costs at each site. In Figure 3, two relations 1 and 2 are referenced by the query part of update transaction, and then both relations are updated according to the update value resulting from the query part.

3.2.3.1. Cost Models for Update Transaction

As mentioned in the previous section, the total cost for executing all query (either OLTP or decision-support) and update transactions against a particular data allocation scheme will determine the goodness of its data allocation scheme, and it is represented as follows:

$$\text{Total Cost} = \sum_k \sum_t F(k,t)Q(k,t) + \sum_u \sum_t F(u,t)U(u,t)$$

Where F(k,t) and F(u,t) are the frequencies of query k originating at site t and update u originating at site t per unit time, and Q(k,t) and U(u,t) are the cost of query k and update u transactions originating at site t. Our objective is to minimize this total cost.

We now define the update transaction cost model. Before describing the cost model, we first introduce one more variable U_i, specifying relations updated by the update transaction. U_i is 1 if relation i is updated by the update transaction; otherwise, it is 0. The update transaction cost is defined as follows:

$$U(u,t) = Q(u,t) +$$

$$\sum_t \sum_p C_{tp} \sum_i U_i X_{it} L_i + \quad (1)$$

$$\sum_t (IO_t \sum_i U_i X_{it} B_i^u + CPU_t \sum_i U_i X_{it} B_i^u) + \qquad (2)$$

$$\sum_t IO_t \sum_i U_i X_{it} L_i \qquad (3)$$

Where, B_i^u is the number of blocks of relation i updated by update u,

L_i is the update value in number of blocks for the relation i, which is the same as the final result from the query part Q(u,t),

IO_t is the I/O cost coefficient (speed) of site t in msec per page (4k bytes),

CPU_t is the CPU cost coefficient (processing speed) of site t in msec per page (4k bytes),

C_{tp} is the communication cost coefficient (channel speed) between site t and site p in msec per page (4k bytes),

X_{it} represents data allocation; relation i is stored at site t.

Note that calculation of query execution time for the query part Q(u,t) of the update transaction is exactly the same as that of the total time model (see below for details). The reason for using the total time model for Q(u,t) is that the update transactions typically occurred in the DEBIT/CREDIT type of transactions in the banking industry, which in general require high throughput. Therefore, the calculation of Q(u,t) is the same as Q(k,t) of total time introduced in Chapter V. In the formula, (1) represents the communication cost for sending the update values (Li) from the update initiation site to all sites that have the copy of the relation being updated; (2) represents I/O cost for reading the required relation into main memory and CPU cost for processing the update; and (3) represents the update cost for writing the updated values back to disk. Calculation of Q(u,t) is as follows:

$$Q(u,t) = \sum_j (LP_j^k + COM_j^k) \qquad (1)$$

$$LP_j^k = \sum_t Y_{jt}^k (IO_t \sum_i Z_{ij}^k B_{ij}^k + CPU_t \sum_i Z_{ij}^k B_{ij}^k) \qquad (2)$$

$$LP_j^k = \sum_t Y_{jt}^k IO_t \sum_m \sum_i \rho_m Z_{ijp[m]}^k B_{ijp[m]}^k + \qquad (3a)$$

$$\sum_t Y_{jt}^k (IO_t \prod_i Z_{ij}^k B_{ij}^k + CPU_t \prod_i Z_{ij}^k B_{ij}^k) \qquad (3b)$$

$$COM_j^k = \sum_m \sum_t \sum_p Y_{jp[m]t}^k Y_{jp}^k C_{tp} (\sum_i Z_{ijp[m]}^k B_{ijp[m]}^k) \qquad (4)$$

where, LP_j^k represents the local processing time of the subquery j of a query k.

COM_j^k represents the communication time of transmitting the input relation(s) to the site at which the subquery j of a query k is being executed.

B_{ij}^k is the number of blocks of relation i accessed by subquery j of query k.

$B_{ijp[m]}^k$ is the size of an input (intermediate) relation where p[m] represents two previous operations of the join operation j: m is 1 for the left previous operation, and 2 for the right previous operation.

ρ_m represents selectivity of the two previous operation (m = 1 or 2), and selectivity refers to the ratio of relation size reduction after an operation.

Y_{jt}^k represents operation allocation and is 1 if subquery j of query k is done at site t; otherwise, it is 0.

$Y_{jp[m]t}^k$ is 1 if the left (m = 1) or right (m = 2) previous operation for join operation j of query k is done at site t; otherwise, it is 0.

Z_{ij}^{k} is 1 if input (or intermediate) relation(s) i is referenced by subquery j of query k.

$Z_{ijp[m]}^{k}$ is 1 if input (intermediate) relation i is referenced by the left (m = 1) or right (m = 2) previous operation for join operation j of query k; otherwise, it is 0.

(1) represents the total query execution time for the query part Q(u,t) of the update transaction and is the sum of all local processing times and communication times. (2) represents the local processing time for the subquery j of the query k when the subqueries are unary operations such as the selection or projection operation. (3a) represents the I/O time in storing the intermediate results of previous operations to the site of the current join operation before the execution of the join. (3b) represents the I/O and CPU processing times for the current join operation. (4) represents the communication time for join operations when either of the (intermediate) relation(s) to be joined is not produced at the site at which the join operation is performed. (4) is also used for the communication time for sending the final result if the final operation is not performed at the query originating site. Since there is only one previous operation for the final operation, we assume that $Z_{ijp[2]}^{k}$ for all i is 0 (also $B_{ijp[2]}^{k}$ = 0).

3.2.4. Cost Model for Load Balancing

We define the unbalanced factor (UBF) as the sum of the absolute deviation of site workloads from the average network load. The objective function for load balancing is then defined to minimize UBF. Minimization of UBF gives a load distribution that has approximately balanced the network load. Note that if the network load among sites is balanced totally (all site have the same workload), the absolute deviation becomes zero. The objective function is defined as follows.

Minimize UBF = $\sum_t |LI_t - LI_{av}| + \sum_t |LC_t - LC_{av}|$

subject to

$$LI_{av} = \frac{1}{N} \sum_t LI_t$$

$$LC_{av} = \frac{1}{N} \sum_t LC_t$$

where LI_t and LC_t represent the I/O and CPU workloads (I/O and CPU times), respectively, at the site t; LI_{av} and LC_{av} represent the average I/O and CPU workloads (I/O and CPU times), respectively, in the entire database; N represents the number of sites. We now define LI_t and LC_t as follows:

(1) For a selection or projection,

$$LI_t = \sum_k F(k,t) \sum_j Y_{jt}^{k}\, IO_t \sum_i Z_{ij}^{k}\, B_{ij}^{k}$$

$$LC_t = \sum_k F(k,t) \sum_j Y_{jt}^{k}\, CPU_t \sum_i Z_{ij}^{k}\, B_{ij}^{k}$$

(2) For a join,

$$LI_t = \sum_k F(k,t) \sum_j Y_{jt}^{k}\, IO_t \sum_m \sum_i \rho_m Z_{ijp[m]}^{k}\, B_{ijp[m]}^{k} +$$

$$\sum_k F(k,t) \sum_j Y_{jt}^{k}\, IO_t \prod_i Z_{ij}^{k}\, B_{ij}^{k}$$

$$LC_t = \sum_k F(k,t) \sum_j Y_{jt}^{k}\, CPU_t \prod_i Z_{ij}^{k}\, B_{ij}^{k}$$

Where, $F(k,t)$ represents the frequency of query k originating at site t, Y_{jt}^{k} represents operation allocation, and is 1 if subquery j of query k is done at site t, otherwise it is 0,

Z_{ij}^{k} is 1 if input (or intermediate) relation(s) i is referenced by subquery j of query k,

$Z^{k}_{ijp[m]}$ is 1 if input (intermediate) relation i is referenced by the left (m = 1) or right (m = 2) previous operation for join operation j of query k, otherwise it is 0, IO_t is the I/O cost coefficient (speed) of site t in msec per page (4k bytes), CPU_t is the CPU cost coefficient (processing speed) of site t in msec per page (4k bytes), B^{k}_{ij} is the number of blocks of relation i accessed by subquery j of query k, $B^{k}_{ijp[m]}$ is the size of an input (intermediate) relation where p[m] represents two previous operations of the join operation j: m is 1 for the left previous operation, and 2 for the right previous operation, and ρ_m represents the selectivity of the two previous operation (m = 1 or 2), and the selectivity refers to the ratio of relation size reduction.

(3) For the update part of an update transaction,

$$LI_t = \sum_u F(u,t)\ IO_t \sum_i U_i X_{it} B_i + \sum_u F(u,t)\ IO_t \sum_i U_i X_{it} L_i$$

$$LC_t = \sum_u F(u,t)\ CPU_t \sum_i U_i X_{it} B_i$$

Where, $F(u,t)$ represents the frequency of update originating at site t, X_{it} represents data allocation; relation i is stored at site t, B^{u}_{i} is the number of blocks of relation i updated by update u, and L_i is the update value in number of blocks for the relation i, which is the same as the final result from the query part of an update transaction.

Note that the query part of an update transaction is the same as (1) and (2) above.

4. Experiments with Integrated Design Method

Interdependency between the problems has been revealed as we describe the design solution for each problem. Figure 1 shows how these problems are interrelated with each other. Our design is based on fragments instead of relations. Thus, we need to address one more issue related to fragmentation. At the vertical fragmentation step, we do not know how the fragments are allocated to sites, which in turn means that we cannot estimate

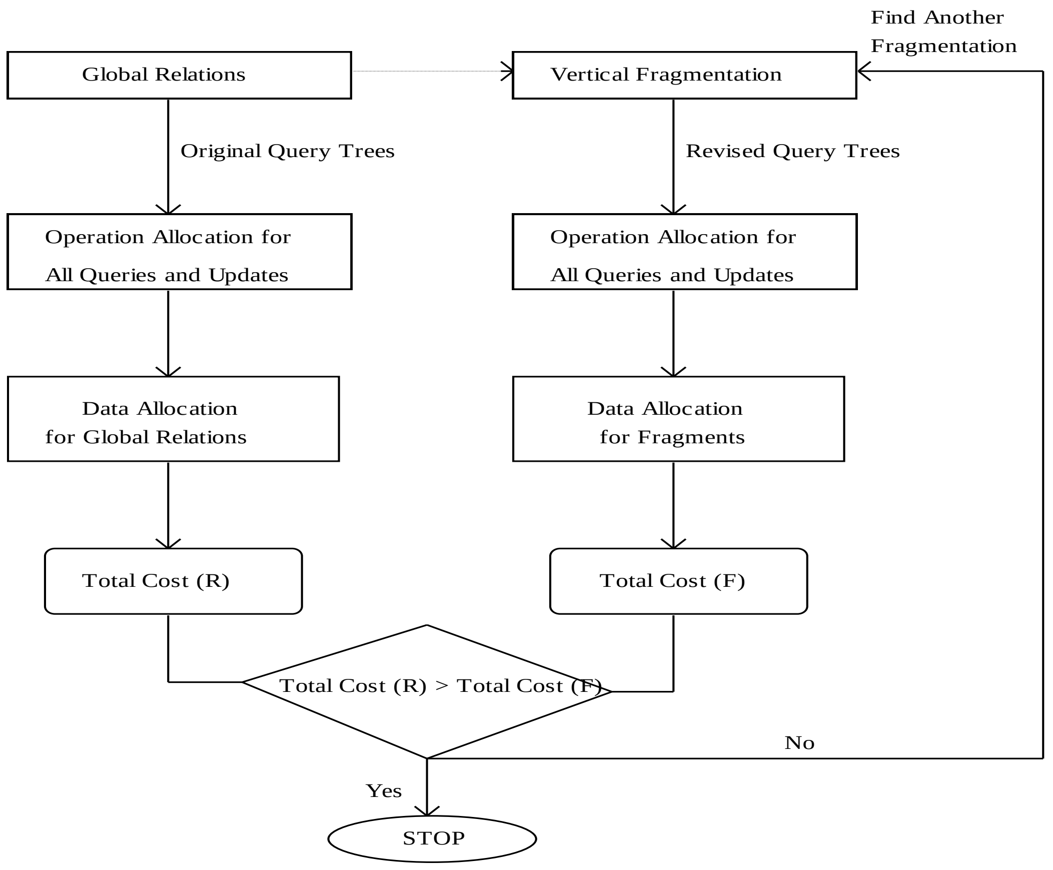
Find Another Fragmentation
Global Relations
Vertical Fragmentation
Original Query Trees
Revised Query Trees
Operation Allocation for All Queries and Updates
Operation Allocation for All Queries and Updates
Data Allocation for Global Relations
Data Allocation for Fragments
Total Cost (R)
Total Cost (F)
Total Cost (R) > Total Cost (F)
No
Yes
STOP

Figure 4. Integrated Design Strategy

Additional join or data transmission costs when transactions require two or more fragments or when the transaction originating site is different from the site where the required fragments are stored. In order to resolve this problem, when a transaction accesses both fragments at each iteration of binary partitioning, the additional cost required for a given partitioning scheme is roughly estimated by adding the penalty cost. However, once the data allocation scheme is determined in accordance with cost minimization operation allocation and load balancing, data transmission cost or additional join cost, instead of the penalty cost, can be calculated for a given partitioning scheme, hence the overall transaction execution cost based on fragments.

Based on the description above, three criteria we use for integrated design method are that (1) two overall distributed database design schemes, one based on global relations and another based on fragments, will be compared; (2) if overall transaction execution cost based on fragments is less than that based on global relations, the overall design procedure will be terminated; and (3) if (2) is not true, then another fragmentation scheme needs to be determined, and the procedure goes back to step (1) above.

Figure 4 shows the revised view of the integrated design steps in Figure 1. As shown in Figure 4, our design strategy is as follows:

(1) Once the dominant query (OLTP and DSS types) and update transactions are selected, the global relations are allocated to the network sites using the genetic algorithms. Let us assume that the total transaction execution cost is Total (R), which is the combination of total times and response times.

(2) For each global relation, the vertical fragmentation genetic algorithm is applied to produce binary fragments first; then it is applied to further partition each fragment until no further fragmentation is indicated based on the transaction profile of each global relation.

The query and update trees are revised according to the fragmentation scheme. Two alternative methods have been suggested for revision [9]:

1. replication of the key attributes at each fragment (Method 1), and

2. use of tuple identifiers (TIDs), which are system-assigned unique values to the tuples of a relation (Method 2).

We employ these two methods and compare their results. In case of Method 1, two cases can occur when query and update trees are revised. First, some operations (subqueries) will access only one fragment, not the whole relation, which results in not only reduced local processing cost but also reduced data transmission cost. Second, some unary operations, however, will need to access two fragments due to binary fragmentation; as a result, an unary operation must be revised and become a join operation. In this case, the local processing cost will be increased and unnecessary data transmission costs will be incurred if the fragments participating in the join operation are allocated to different sites.

In case of Method 2, when an unary operation needs to access two fragments, it performs its local operation on the first fragment first, then the necessary tuple identifiers are obtained from the first fragment. The tuples in the second fragment are then obtained by using these tuple identifiers. If the site at which the second fragment is stored is different from that of the first fragment, the costs are calculated as follows:

1. the cost of sending TIDs to the site of the second fragment,

2. the costs of I/O and CPU processing on the second fragments based on TIDs, and

3. the cost of sending the tuples selected from the second fragment to the site of the first fragment.

Note that if the first fragment and the second fragment are stored at the same site, then the costs 1 and 3 are not incurred.

(4) Based on the revised query and update trees, the fragments are allocated to the network sites, and let us assume that the total transaction execution cost is Total (F).

(5) If Total (F) is less than Total (R), then we stop the design steps since the total cost is reduced because of vertical fragmentation.

(6) If Total (F) is more than Total (R), then the allocation based on the fragments is worse than that of global relations. In this case, we need to find another fragmentation scheme and compare the result again. Note that since the vertical fragmentation genetic algorithm produces many alternatives in its population pool, we may choose the second best fragmentation scheme from the pool for one of the global relations, then repeat the steps (3) to (5). If after several attempts all vertical fragmentation schemes fail to reduce the total cost to less than that of using the global relations, we may conclude that it is better not to use vertical fragments for this particular database, probably due to the query and update transaction patterns. But it is likely that since our vertical fragmentation scheme produces the best possible fragmentation, which not only reduces the local processing cost but also minimizes accessing binary fragments-reducing join operations, its total cost will be reduced to less than that of using the global relations, even though not guaranteed.

In the following sections, we illustrate these steps based on one fictitious database.

4.1. Global Database Specification

In order to illustrate the integrated design method, we will use one fictitious distributed database system which consists of five database sites. A global database schema is assumed to be developed and consists of seven relations. The size of each relation is as follows: relation1 is 4850, 2 is 3500, 3 is 2500, 4 is 3000, 5 is 3750, 6 is 1500, and 7 is

508. The size of a relation is measured in data page blocks and the size of data page is assumed to be 4k bytes. The design problem at hand is that these relations will be fragmented and allocated among five sites according to the user transaction pattern.

We assume that the dominant transactions are selected already, and the set of dominant query and update transactions to be executed by the proposed database system are summarized in Table 1. Tables 1 also describes the relations required by each transaction, the frequency, and the transaction originating site. A "1" in the tables indicates a particular relation needed by a transaction. We assume that query trees (query execution order) for each of the query transactions is derived through a global (or local) query optimizer. We also assume that queries 1-10 are left deep query tree types for the total cost minimization while queries 11-18 are the bushy query tree types for the response time minimization. The cost coefficients assumed in this research are as shown in Table 2.

Table 1. Retrieval and Update Transactions on Global Relations

Retrieval (Query) Transactions																		
Relation	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1				1	1			1	1	1	1		1	1	1	1	1	1
2	1		1				1	1	1	1	1	1	1	1	1		1	1
3	1	1			1	1	1	1	1	1	1	1			1	1	1	1
4		1	1	1	1	1	1	1	1	1	1	1	1			1	1	1
5		1		1		1			1				1	1		1	1	1
6						1	1	1	1			1	1	1	1	1	1	1
7	1	1		1		1			1	1	1			1	1	1	1	1
Frequency	10	10	7	7	5	5	5	5	2	2	2	2	2	2	1	1	1	1
Query Site	4	3	4	1	3	2	2	4	5	1	4	2	4	2	3	5	1	3

Update Transactions										
	Query Part					Update Part				
Relation	1	2	3	4	5	1	2	3	4	5
1	1	1	1	1	1		1			
2	1									1
3		1				1				
4			1					1		
5				1					1	
6			1		1					1
7		1		1		1				
Frequency:	3	2	2	1	1					
Site:	4	1	2	3	5					

Table 2. Cost Coefficients for I/O, CPU, and Communication

		Site				
	Site	1	2	3	4	5
Communication Speed	1	0	1.2	1.0	0.8	1.2
	2	1.2	0	1.0	1.2	1.0
	3	1.0	1.0	0	1.2	1.2
	4	0.8	1.2	1.2	0	1.0
	5	1.2	1.0	1.2	1.0	0
I/O Speed		2.5	2.0	2.5	2.0	2.2
CPU Speed		0.1	0.1	0.1	0.1	0.1

4.2. Vertical Fragmentation

The genetic algorithm is applied to produce binary fragments first, and then applied to further partition each of them until no further fragmentation is indicated. The transaction profile for each relation is assumed to be generated and will be used as the input to the vertical fragmentation genetic algorithm. Each transaction profile is used to determine whether the fragmentation of a relation results in the reduction of the total transaction execution cost. When applying the vertical fragmentation genetic algorithm, five relations (relation 2, 3, 4, 5, and 6) do not produce any good fragment, and so each relation itself will be used as the unit of allocation, and so the vertical fragmentation is applied to the relations 1 and 7 only.

The transaction profiles for relations 1 and 7 are described in Table 3 and Table 4, respectively. In Table 5 it is assumed that each "Tx" represents the particular set of queries or updates which have the same attribute access pattern. Each transaction represents the following queries:

Tx 1: Query 9, 10
Tx 2: Query 15, 16, 17
Tx 3: Query 14, Update 5
Tx 4: Query 13, 18, Update 4
Tx 5: Update 1
Tx 6: Query 4, 5, 8, 11

Table 3. Transaction Profile for Relation 1

	Attributes																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tx 1	1	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Tx 2	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Tx 3	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
Tx 4	0	0	0	0	0	0	1	0	0	0	1	1	1	1	1	0	0	0	0	0
Tx 5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Tx 6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
Length:	8	8	8	8	4	8	8	12	20	22	4	8	6	5	3	30	12	8	6	6

Restrict Attribute:	1	1	8	7	1	15
Selectivity :	.001	.001	.001	.001	.001	.001
Frequency :	4	3	3	4	3	23

Table 4. Transaction Profile for Relation 7

	Attributes																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tx 1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
Tx 2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
Tx 4	0	1	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0
Tx 6	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	1	0	0
Tx 9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1
Tx 10	0	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0
Tx 11	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0
Tx 14	1	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0
Tx 15	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1	1
Tx 16	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0	0
Tx 17	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	1	1	1	0	0
Tx 18	1	1	1	0	1	1	0	0	1	0	0	1	1	0	0	0	0	0	0	0
Up 1	1	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Up 4	0	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	0	1
Up 5	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Length:	8	8	8	8	4	8	8	8	4	4	4	8	6	5	30	30	12	8	16	16

Restrict Attr.:	3	15	2	3	15	2	1	1	4	3	8	1	1	2	1
Selectivity:	.005	.005	.005	.005	.0015	.0015	.0015	.0015	.001	.001	.001	.001	.001	.0005	.0005
Frequency:	10	10	7	5	2	2	2	2	1	1	1	1	4	1	1

The cardinality of relation 1 is 100,000 and that of relation 7 is 10,000. It is assumed that the length of tuple identifier is 4 bytes and the size of index page for unclustered index scan is 10. For all relations, it is assumed that attribute 1 is taken to be the clustering attribute, and indexes are available on all restrict attributes. The number of scans per run for all transactions is assumed to be 1.

The genetic algorithm is then applied to each fragment, and fragment 1 is further partitioned into two fragments as follows:

Fragment 1-1: 1, 2, 3, 4, 5, 6, 8, 9, 10

Fragment 1-2: 7, 11, 12, 13, 14

Any further fragmentation gives no cost reduction, and so three-way fragmentation is the best solution in this case. As a result of applying vertical fragmentation to relation 1, each query accesses different fragment(s), and after renumbering the fragments 1-1, 1-2, and 2 as 1, 2, and 3 respectively, it is summarized as follows:

- Query 9, 10: Fragments 1 and 2
- Query 14, 15, 16, 17, Update 5: Fragment 1
- Query 13, 18, Update 4: Fragments 2 and 3
- Update 1: Fragments 1 and 3
- Query 4, 5, 8, 11, Update 1: Fragment 3

The application of genetic algorithm to relation 7 results in two fragments as follows:

Fragment 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 17, 18

Fragment 2: 15, 16, 19, 20

As a result of applying vertical fragmentation on relation 7, each query accesses different fragment(s) and is summarized as follows:

- Query 1, 4, 6, 11, 14, 16, 18, Update 2, 5: Fragment 1
- Query 9: Fragment 2
- Query 2, 10, 15, 17, Update 4: Fragments 1 and 2

We assume that the approximate size of three fragments from relation 1 is 2450, 775, and 1625, respectively and that of two fragments from relation 7 is 278 and 230, respectively.

4.3. Data Allocation with Operation Allocation and Load Balancing

The allocation of fragments is achieved through an iterative procedure between operation allocation and data allocation with or without load balancing. Since we have determined the unit of allocation in the previous phase, we revise the query and update transactions in terms of fragments, and the query execution order for each query is also revised in terms of fragments. Table 5 describes the fragments required by each transaction, the frequency, and the transaction origination site.

4.3.1. Allocation of Global Relations

As a result of the genetic algorithm, data allocation for the global relations is obtained as follows: relation 1 is allocated to site 3, relation 2 is allocated to site 1, relation 3 is allocated to site 5, relation 4 is allocated to site 4, relation 5 is allocated to site 2, relation 6 is allocated to site 2 and 4, and relation 7 is allocated to site 4. The total query and update execution cost is 1,782,185 time units, and the unbalanced factor is 1,255,780.

4.3.2. Allocation of Fragments

By using Method 1, data allocation for the fragments is obtained as follows: fragment 1 is allocated to site 1 and 5, fragment 2 is allocated to site 3, fragment 3 is allocated to site 4, fragment 4 is allocated to site 2, fragment 5 is allocated to site 1 and 4, fragment 6 is allocated to site 5, fragment 7 is allocated to site 4, fragment 8 is allocated to site 3, fragment 9 is allocated to site 5, and fragment 10 is allocated to site 1. The total query and update execution cost is 1,249,792 time units, and the unbalanced factor is 1,038,636. Using Method 2 results in data allocation as follows: fragment 1 is allocated to site 2 and

Table 5. Query and Update Transactions on Fragments

Retrieval (Query) Transactions

Fragment	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1									1	1				1	1	1	1	
2									1	1			1					1
3				1	1			1			1		1					1
4	1		1				1	1	1	1	1	1		1	1		1	1
5	1	1			1	1	1	1		1		1			1	1	1	1
6		1	1	1	1	1	1	1	1	1	1	1	1			1		1
7			1		1		1						1	1		1	1	1
8						1	1	1	1			1	1	1	1	1	1	
9	1	1								1					1		1	1
10	1	1		1		1			1	1	1			1	1	1	1	
Frequency:	10	10	7	7	5	5	5	5	2	2	2	2	2	2	1	1	1	1
Query Site:	4	3	4	1	3	2	2	4	5	1	4	2	4	2	3	5	1	3

Update transactions

	Query Part					Update Part				
Fragment	1	2	3	4	5	1	2	3	4	5
1	1		1		1		1			
2				1			1			
3	1	1		1						
4	1						1			1
5			1							
6				1					1	
7					1					1
8			1		1					1
9				1	1					
10		1		1	1					
Frequency:	3	2	2	1	1					
Site:	4	1	2	3	5					

5, fragment 2 is allocated to site 1and 3, fragment 3 is allocated to site 4, fragment 4 is 2allocated to site 2, fragment 5 is allocated to site 1 and 4, fragment 6 is allocated to site 5, fragment 7 is allocated to site 1, fragment 8 is allocated to site 4, fragment 9 is allocated to site 1 and 5, and fragment 10 is allocated to site 2. The total cost is 1,070,299, and its unbalanced factor is 818,752. As a result, the total cost of data allocation with vertical fragmentation has been reduced by 532,393 for Method 1 and 711,886 for Method 2, respectively. Note that the execution times for some queries are increased as the results of vertical fragmentation. However, overall total cost is reduced, and so the design steps are stopped.

5. Conclusions

Several interrelated issues are involved in the design of distributed database systems. The complexity of the individual problems, as well as the interdependencies among the problems, makes the entire design process computationally intractable. In this paper, we have proposed a new solution method for partitioning relations, allocating partitioned fragments among the sites of a network, and allocating database operations for distributed query optimization. We have proposed as a solution methodology the genetic algorithm and successfully demonstrated its usefulness for providing an efficient search method for the problems addressed in this paper.

We have developed an integrated cost models which differ from previous studies in that we drop the assumption that the query transaction accesses only one relation (or fragment) independently of the other relations (or fragments). In our cost models, a query transaction is modeled by a query tree which represents a set of subqueries together with their precedence relationship. In addition, the query transactions are classified into two groups: (1) OLTP (on-line transaction processing) and (2) DSS (decision support system) types. For OLTP types of query transaction, the left deep query tree is employed since it provides the better query execution order in terms of minimizing the total time. For DSS types of query transactions, the bushy query tree is employed since it provides the better query execution order in terms of minimizing the response time. This research is one of a few studies which use as the objective function a linear combination of total time and response time. Furthermore, we integrate load balancing and total cost (linear combination of total and response time) minimization into operation allocation and data allocation. Especially, the integration of load balancing to data allocation may be the first attempt, to the best of our knowledge.

In summary, this paper essentially introduced some new approaches to solve computationally complex problems encountered in the design of distributed database systems. Such techniques are in great demand as computer and communication technologies are advanced in faster speed.

Acknowledgement

This work was supported by a 2017 research grant from Youngsan University, Republic of Korea.

References

[1] C. Cheng, W. Lee, and K. Wong, “Genetic algorithm-based clustering approach for database partitioning”, IEEE Transactions on Systems, Man, and Cybernetics, vol. 32, no. 3, **(2002)**, pp. 215-230.
[2] J. Du, R. Alhajj, and K. Barker, “Genetic algorithms based approach to database vertical partitioning”, Journal of Intelligent Information Systems, vol. 26, no. 2, **(2006)**, pp. 167-183.
[3] X. Gu, W. Lin, and V. Bharadwaj, “Practically realizable efficient data allocation and replication strategies for distributed databases with buffer constraints”, IEEE Transactions on Parallel and Distributed Systems, vol. 17, no. 9, **(2006)**, pp. 1001-1013.
[4] I. Hababeh, R. Omer, and B. Nicholas, “A high-performance computing method for data allocation in distributed database systems”, Journal of Supercomputing, vol. 39, no. 1, **(2007)**, pp. 3-18.
[5] Y. Jiang and J. Jiang, “Contextual resource negotiation-based task allocation and load balancing in complex software systems”, IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 5, **(2009)**, pp. 641-653.
[6] D. Kossmann, “The state of the art in distributed query processing”, ACM Computing Surveys, vol. 32, no. 4, **(2000)**, pp. 422-469.
[7] M. Lin, “An optimal workload-based allocation approach for multidisk databases”, Data & Knowledge Engineering, vol. 68, no. 5, **(2009)**, pp. 499-508.
[8] S. Menon, “Allocating fragments in distributed databases”, IEEE Transactions on Parallel & Distributed Systems, vol. 16, no 7, **(2005)**, pp. 577-585.
[9] M. Ozsu and P. Valduriez, “Principles of Distributed Database Systems”, Cliffs, N. J., Prentice-Hall, Englewood, **(2011)**.
[10] E. Sevince and A. Cosar, “An evolutionary genetic algorithm for optimization of distributed database queries”, Computer Journal, vol. 54, no. 5, **(2011)**, pp. 717-725.
[11] S. Song and N. Gorla, “Genetic algorithm for vertical fragmentation and access path selection”, Computer Journal, vol. 43, no. 1, **(2000)**, pp. 81-93.
[12] S. Song, “Design of distributed database systems: an iterative genetic algorithm,” Journal of Intelligent Information Systems”, vol. 45, no. 1, **(2015)**, pp. 29-59.
[13] A. Tamhankar and S. Ram, “Database fragmentation and allocation: an integrated methodology and case study”, IEEE Transactions on Systems, Man, and Cybernetics: Part A., vol. 28, no. 3, **(1998)**, pp. 288-295.
[14] A. Verma and M. Tamhankar, “Reliability-based optimal task-allocation in distributed-database management systems”, IEEE Transactions on Reliability, vol. 46, no. 4, **(1997)**. pp. 452-459.
[15] J. Wang and K. Jea, “A near-optimal database allocation and replication strategies for distributed databases with buffer constraints”, Information Sciences, vol. 179, no. 21, **(2009)**, pp. 3772-3790.

Author

Sukkyu Song, Ph. D, Professor, Youngsan University, Manager, Posdate Co., Seoul, Korea.