Load Balancing in Distributed Query Processing*

Chengwen Liu and I-Ping Chu
Computer Science, DePaul University, Chicago, USA
liu,chu@cs.depaul.edu

Abstract

Query processing is a very important issue in distributed databases. Many algorithms have been proposed to process distributed queries efficiently. However, most of the algorithms use oversimplified cost models and ignore the impact of work load generated by other applications. As a result, load balancing is difficult to achieve in a real environment. In this paper, we provide an adaptive scheme to do load balancing effectively. The scheme takes into account an environment in which the load at different sites varies. The Partition and Replicate Strategy algorithm is used to explain how to achieve load balancing in a multi-user environment. The scheme also has learning capability such that the parameters of cost estimation functions can be adaptively adjusted as the environment changes.

1 Introduction

In a distributed database environment, data is stored at different sites connected through a network. One of the important problems in distributed database management is the efficient processing of queries. In [1, 18], an experimental distributed query processing system built on top of local DBMSs was presented. One of the algorithms used for query optimization in [1, 18] is the Partition and Replicate Strategy (PRS) algorithm[20]. Validation and thorough analysis of the performance of the PRS algorithm was given in [12]. It was shown that the PRS strategy outperforms single site processing in a realistic environment and the PRS algorithm chooses the correct set of processing sites. However, load balancing among the chosen processing sites is not achieved. In [11], we identify the factors that cause load imbalance, point out the difficulties for achieving load balancing and provide detailed cost models to get more accurate cost estimates for communications and local processing. However, inaccuracies still exist. For example, if the amount of data involved in processing a join exceeds certain value, then there is a big jump in the processing cost [15]. It seems that the amount of buffer space available in the main memory for processing the query is critical.

In addition, in a multi-user environment, work load generated by other users has a very important impact on the execution time of a query. Thus, it is desirable to do load balancing by assigning smaller amount of processing to a heavily loaded site. In this paper, we propose an adaptive learning scheme to do load balancing effectively in a multi-user environment.

The remainder of this paper is organized as follows. In Section 2, we give an overview of our load balancing scheme. In Section 3, we give a summary of the cost models proposed in [11]. In Section 4, we discuss load balancing in multi-user environment. In Section 5, we discuss the adaptation of cost functions. In Section 6, we describe how to apply the load balancing techniques to the PRS algorithm [20]. Finally, we conclude in Section 7.

2 Load Balancing and Query Optimization

There are several ways to integrate load balancing into the query optimization and execution mechanisms of a distributed database system. One approach would be to integrate a dynamic query allocation algorithm like the one given in [3] into a distributed query processing algorithm, letting the query processing algorithm select the next subquery to execute and the dynamic query allocation algorithm select from among the candidate sites for executing the query. The criteria used for selecting the processing site is "balance the number of queries", where a newly arrived query is routed to the site with fewest queries, or "balance the number of queries by demands", where a newly arrived query is classified as being either CPU bound or I/O bound and is routed to the site with the fewest queries of the

*Research supported in part by Trilogy Technologies, Inc.

Proceedings of the Fourth International Conference on
Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

same class. The problem with this approach is that only the number of queries is considered. However, in a general purpose computing environment, there may be other type of users that compete for the same resources. The second possible approach would be to integrate load information into the query optimization algorithms to select the optimal plan by taking into consideration of the impacts of multiple users. In other words, cost estimation for the optimization algorithms are obtained by taking into consideration of the system load information [17]. The problem with this approach is that it would preclude query compilation. As pointed out in [4], this leads to an unacceptable runtime overhead for query optimization. This problem can be relieved by compiling a query into a collection of alternative plans instead of a single plan, associating a load constraint with each plan to specify the system load conditions under which it should be used [4]. This approach is attractive in terms of runtime overhead, but it complicates query compilation since various possible system load conditions would have to be considered during query compilation. In addition, the number of alternative plans that can be generated will be limited by storage space and query complexity. Yet another approach is to use load information to dynamically modify execution plans such that better response times can be achieved. In this paper, we propose a learning scheme as shown in Figure 1 to do load balancing effectively for distributed query processing.

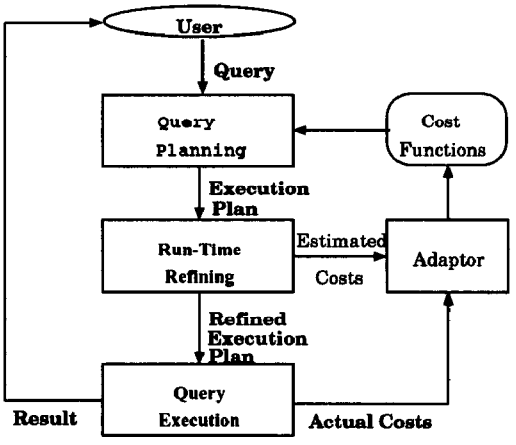

Figure 1: Learning scheme for load balancing

The learning scheme consists of four major components. The first component is called *query planning* which is actually the query optimizer as described in [13]. However, the following modifications are made:

1. Instead of using fixed cost functions, costs of operations are estimated by using the dynamically modified parameters.
2. Instead of generating a single execution plan, a set of plans, which are comparable to the optimal plan, are generated. A execution plan is comparable to the optimal plan if its estimated total cost is not significantly greater than that of the optimal plan.
3. The set of plans together with the cost estimation results are passed to the next phase.

The second component is called *run-time refining* which takes as input the set of execution plans generated by the query planing component and uses the load information at each of the processing sites to modify those plans such that load balancing can be achieved in a multi-user environment. Then the execution plan which has the lowest total cost will be chosen and given to the next component, *query execution*, for execution. During execution, execution times of the operations at each processing site are collected and passed to the *Adaptor*, that decides which parameters of the cost model need to be modified to reflect the actual cost so that cost estimations for later queries can be more accurate.

Adaptive cost estimation is much less popular in the literature as compared to static cost estimation (see survey [9]). As pointed out in [17], static cost estimation which can be highly inaccurate [5] should only be used to initialize the cost estimation formulas. An advantage of our approach is that it simplifies query optimization, since query planning can be performed without considering load information [4].

3 Cost Models

In a distributed database environment, the analysis of costs for query evaluation includes estimations of (1) the size of a relation resulting from a join, selection or projection, (2) data transfer rate between sites, and (3) the local processing costs. Relation size estimation has been studied in fairly detail in the literature[6, 8, 10]. Simple cost models for estimating communications costs and local processing costs have been given in [19, 20]. However, our experience with the PRS algorithm indicates that these models produce incorrect estimates and as a result, load balance can not be achieved. In order to achieve better load balancing, sufficient detailed cost models are needed to get more accurate estimates. Based on our experience with distributed query processing and realistic experimental results, we provide detailed cost models for primitive operations of distributed query processing [11].

3.1 Communications Cost

Transmitting data across sites consists of three actions: (a) retrieve data from secondary memory at the sending site, (b) send data across the communications network, and (c) store data into secondary memory at the receiving site. The costs of these three actions can be given by the following functions

$$T_{ret}(M) = R_0 + R_1 \times M$$
$$T_{net}(M) = N_0 + N_1 \times M$$
$$T_{sto}(M) = S_0 + S_1 \times M$$

where M is the number of units of data (a unit can be a page/packet) sent across the network, symbols with subscript 0 are the start-up costs and symbols with subscript 1 are the proportional constants per unit. The parameters R_0, R_1, S_0, S_1, N_0 and N_1 can be determined experimentally [1].

3.2 Local Processing Costs

Our experience with distributed query processing in a fast local network indicates that local processing costs are significant and should not be ignored. In other words, in a local network, communications cost is not the dominant factor. Estimation of local processing cost include estimating the costs for joins, projections, selections, unions and partitions.[1] Because of space limitation, we will only present the cost functions for selection. Cost functions for the other operations can be found in [11].

When a distributed query involves selections/projections, the selections/projections can be performed in order to reduce the size of data to be transferred across the network [13]. These operations are called local reduction. The cost of local reduction depends on the selectivity factor (how many result tuples are produced by the query) and the storage organization (access method) [2]. In general, we have the following 4 cases: (1) the relation is stored by hashing on the selection attribute; (2) the relation is sorted and a primary index has been built on the selection attribute; (3) there is a secondary index on the selection attribute; and (4) there is no fast access path.

Our experimental results [14] show that the cost of a selection query is a linear function of the number of tuples selected provided the file organization is fixed. Thus, the following functions can be used to estimate the cost of selection queries:

$$T_{lr}(R) = \begin{cases} h_0 \times X + h_1 \times Y & \text{if hashed ,} \\ p_0 \times X + p_1 \times Y & \text{if primary index,} \\ s_0 \times X + s_1 \times Y & \text{if secondary index,} \\ n_0 \times X + n_1 \times Y & \text{otherwise.} \end{cases}$$

where R is the referenced relation, X and Y are the sizes of R before and after the local reduction respectively. The symbols with subscript 0 (h_0, p_0, s_0, n_0) represent the cost of selecting zero tuple from the relation under different storage organizations. For example, in case 4, $n_0 \times X$ is the time needed to scan the whole relation. However, if the relation is indexed, the parameters (s_0, p_0) will represent the overhead of searching the index. The parameters with subscript 1 (h_1, p_1, s_1, n_1) are proportional constants per selected tuple. These parameters can be very easily determined by experiments [11].

4 Load balancing in Multi-user environment

The *run-time refining* component in Figure 1 is used to achieve load balancing in a multi-user environment. The input to this process is a set of execution plans generated by the query planning process based on the cost estimations in a single user environment. The execution plans will be refined by considering the workload of each processing site. First, the system sends a message to each processing site requesting the workload statistics, such as the number of users in the run queue, the number of blocked users, the CPU status and the I/O status. For example, the UNIX system can provide the utilization of the CPU, the utilization of the I/O devices and the number of users in the run queue and the number of users blocked for I/O. A user is in the run queue if it is waiting for the CPU and a user is blocked if it is waiting for the I/O device. Assume that μ_{cpu}, μ_{io}, b_{cpu} and b_{io} are the CPU utilization, I/O utilization and the number of queued users and number of blocked users, respectively. Then we can define the state of a site as one of the following:

1. I/O bound – μ_{io} is close to 100% and μ_{cpu} is not close to 100%,
2. CPU bound – μ_{cpu} is close to 100% and μ_{io} is not close to 100%,
3. both I/O bound and CPU bound – both μ_{io} and μ_{cpu} are close to 100%,
4. neither I/O bound nor CPU bound – both μ_{io} and μ_{cpu} are not close to 100%.

[1] Partition refers to the operation to partition a relation into horizontal fragments.

Queries can be classified into a number of query types (e.g., select, join, union, partition, and etc.). Queries can also be classified into different classes based on their usage of the CPU and I/O [3, 17]. Assume T_{tot}, T_{cpu} and T_{io} are the total elapsed time, the CPU time and the I/O time[2] of a query respectively in a single user environment, then the following criteria can be used to classify the query:

1. I/O intensive – $T_{io}/T_{tot} > \rho$, where ρ is a value close to 1. In other words, I/O time is much greater than the CPU time required to process this query.

2. CPU intensive – $T_{cpu}/T_{tot} > \rho$, the query requires significant amount of CPU time with small usage of I/O.

3. both I/O and CPU intensive – $T_{io}/T_{tot} > \rho$ and $T_{cpu}/T_{tot} > \rho$, this case arises only if accessing a block and processing a block take approximately the same time and the implementation allows these two operations to overlap.

4. neither I/O intensive nor CPU intensive – $T_{io}/T_{tot} \leq \rho$ and $T_{cpu}/T_{tot} \leq \rho$.

Based on the above definition, a given query can only be classified by running it in a single user environment and collecting the necessary information. This is clearly impractical. However, we can initially classify a query based on its query type and the type of access paths available. For example, partition is likely to be I/O intensive. Since our system has learning ability, classification of queries can then be determined based on the actual costs collected during their execution.

Each execution plan passed from the query planning phase consists of a set of query units (a query unit may be a selection, a projection, a join, a relation migration, etc.) for each site. Associated with each query unit are the following attributes: (1) QT – an integer which represent the query type (selection, projection, join, union, partition, etc.) (2) QC – a number which represents the query class (one of the classes described above), (3) AP – access path, and (4) EC – estimated cost in single user environment. These information, together with the actual costs collected in the query execution process, will be used by the *Adaptor* to adjust the cost model parameters to reflect the actual costs. Only the estimated cost and the query class are used for the purpose of refining.

Since the execution plans are generated without considering the workload at the processing sites, load imbalance (total time at a site with heavy load will be much larger than that of a site with very light load) may be observed in a multi-user environment. In order to achieve load balancing, we modify the cost of each query unit by taking into consideration of the workload at each individual processing site and then balance the total times at all the processing sites by assigning some of the work previously allocated to a site with heavy load to a different site having lighter load.

If a query unit of class QC is assigned to a site in state S, then its processing cost in the multi-user environment will be

$$MEC = f(QC, S) \times EC$$

where EC is the estimated cost in the single user mode and $f()$ is a function which can be determined as follows.

Assume that the workload caused by other users remains the same throughout the execution of the given query and the system uses round-robin scheduling for both the CPU jobs and the I/O jobs. Then, the impacts on I/O and CPU times by multiple users can be represented by the following formulas:

$$MT_{io} = (b_{io} + 1) \times T_{io} = (b_{io} + 1)\rho_{io} \times T_{tot}$$
$$MT_{cpu} = (b_{cpu} + 1) \times T_{cpu} = (b_{cpu} + 1)\rho_{cpu} \times T_{tot}$$

where MT_{io} and MT_{cpu} are the estimated I/O and CPU costs in the multi-user environment with the given workload, and ρ_{io} and ρ_{cpu} are the percentages of the I/O and CPU costs to total cost, i.e., $\rho_{io} = T_{io}/T_{tot}$ and $\rho_{cpu} = T_{cpu}/T_{tot}$. In [14], we have shown that the total time of a query can be expressed as

$$T_{tot} = T_{io} + T_{cpu} - \alpha \times min\{T_{io}, T_{cpu}\}$$

where α is the overlap factor between the CPU and the I/O. If we further assume that α is independent of the number of users, then we have

$$\begin{aligned} MEC &= MT_{io} + MT_{cpu} - \\ &\quad \alpha \times min\{MT_{io}, MT_{cpu}\} \\ &= ((b_{io} + 1)\rho_{io} + (b_{cpu} + 1)\rho_{cpu} - \\ &\quad \alpha \times min\{b_{io}\rho_{io}, b_{cpu}\rho_{cpu}\}) \times EC \end{aligned}$$

where $EC = T_{tot}$. Thus,

$$\begin{aligned} f() &= (b_{io} + 1)\rho_{io} + (b_{cpu} + 1)\rho_{cpu} \\ &\quad -\alpha \times min\{b_{io}\rho_{io}, b_{cpu}\rho_{cpu}\} \end{aligned}$$

Note that if the system is not I/O bound, then $b_{io} = \mu_{io}$. Similarly, if the system is not CPU bound, then $b_{cpu} = \mu_{cpu}$. The relative CPU and I/O costs (ρ_{cpu}

[2] Usually the I/O time is defined as the product of the average block access time and the number of I/Os [7, 14].

and ρ_{io}) can usually be estimated by the information provided by the optimizer since many optimization algorithms can provide the estimations of CPU and I/O costs. In some systems, if these information are not provided, ρ_{cpu} and ρ_{io} can be determined by the query class. For example, a CPU intensive query has $\rho_{cpu} \approx 1$ and $\rho_{io} \approx 0$. In this case, the value of α is not important. However, for both CPU and I/O intensive query, $\rho_{cpu} \approx 1$ and $\rho_{io} \approx 1$. In this case, α is also close to 1. For a query which is neither CPU intensive and nor I/O intensive, the value of α is likely to be 0. Since our system has learning ability, a better approach for estimating the values of α is to learn from the actual CPU cost, I/O cost and the total cost.

In the above, we have discussed how the response time of a query can be estimated in a multi-user environment by using the workload information (number of queued users, number of blocked users, CPU utilization and I/O utilization) of the system. However, these information may not be available on some systems. As a complement, the following model can be used to estimate the effect of time sharing [7]:

$$MEC = \beta_0 U^{\beta_1} \times EC$$

where U is the number of users logged on to the system and β_0 and β_1 are regression parameters which can be determined experimentally [7]. Since not all of the U users are requesting or using the CPU or the I/O device, $(\beta_0 U^{\beta_1})$ incorporates an estimate of the percentage of users actually using or requesting the CPU or I/O device.

5 Adaptation of Cost Functions

In Section 4, we have discussed how a given execution plan can be refined to achieve load balance in a multi-user environment by taking into consideration of the workload of each processing site. Clearly, we have assumed that the given plan is load balanced for a single user environment, i.e., each site has approximately the same execution time if there is no other user on the system. However, because of inaccuracy of cost estimation, load balance may not be achieved even there are no other users on the system. Therefore, some learning capability should be added to improve cost estimations for query execution strategies (refer to Figure 1). In the following, we discuss how the cost functions can be adapted to reflect the actual costs of queries.

The input to the *Adaptor* is a set of query units with their estimated costs and corresponding actual costs collected after query execution. A query unit can be any one of the following: (1) local reduction — perform selection and/or projection on a relation to reduce its size; (2) partition — partition a relation into a number of fragments; (3) relation/fragment migration — transfer a relation/fragment from one site to another site; (4) union — combine two or more fragments of a relation to form a whole relation; (5) join with or without selection or projection — this is the query executed in the subquery processing phase [13]. For each query unit, the adaptor will adjust its corresponding cost function by comparing the estimated cost and the actual cost.

We first consider relation migration. We believe the technique described in [17] can be applied. Since data transfer rate for different lines may be different, we maintain a cost function for each pair of sites. The cost of transferring a relation consisting of X blocks from one site to another site can be expressed as

$$T_{xfr}(X) = c_0 + c_1 \times X$$

where c_0 is the time to establish a connection between the sender and the receiver and c_1 represents the reciprocal of the speed of the transmission (in time units per block). Initially, each line can be estimated to have the same data transfer rate (the same values of c_0 and c_1). If the actual data transfer cost is *significantly* below or above the estimated cost, then data transfer rate is increased or decreased by a value proportional to the difference between the actual and the estimated costs.

More precisely, if the actual cost of establishing a connection is ac_0 and the actual cost of transmitting X block of data is $ac_1 \times X$, then c_0 and c_1 can be adjusted as the following:

$$c_0 = c_0 + \alpha \times (ac_0 - c_0) \quad \text{if } \frac{|ac_0 - c_0|}{ac_0} > \kappa$$
$$c_1 = c_1 + \alpha \times (ac_1 - c_1) \quad \text{if } \frac{|ac_1 - c_0|}{ac_1} > \kappa$$

where $0 < \alpha < 1$ and $0 < \kappa < 1$. The value of α controls the speed of adaptation and the value of κ prevents too frequent adaptive changes.

Now we consider local processing cost. Different sites may have different processing speeds. This is handled by introducing a relative processing speed for each site [19]. For example, the slowest site may be assigned the number 1. If a site is x times as fast as the slowest processing site, then its relative processing speed is x. Thus, if a query is processed at a site, the cost is to divide the cost function by the relative processing speed of the site. In this case, a single cost function is used for all the sites. However, our experience indicates that this approach introduces significant errors. The relative processing speed of a site is query dependent since the speedups of the CPU and the I/O devices

are usually different and the requirement of CPU and I/O are also different for different queries. Therefore, cost functions should be maintained for each site and each type of query. These functions are dynamically changed by the adaptor to reflect the actual costs.

For a given query unit, assume the actual CPU time is T_{cpu}, the actual I/O time is T_{io} and the overlap between the CPU and the I/O is α. Then the total time in single user mode will be $T_{io} + T_{cpu} - \alpha \times min\{T_{io}, T_{cpu}\}$. The overlap factor α can be determined by the query class. For example, if the query is I/O intensive or CPU intensive, then α will be close to 1; otherwise if the query is neither CPU intensive nor I/O intensive, then α will be close to 0. If we use a select/project query as an example, the following formula can be obtained:

$$T_{io} + T_{cpu} - \alpha \times min\{T_{io}, T_{cpu}\} = l_0 \times X + l_1 \times Y$$

where l_0 and l_1 are constants, X is the size of the relation before the operation and Y the size of the result. Note that l_0 is one of h_0, p_0, s_0 or n_0 depending on the storage organization of the referenced relation. Similarly, $l_1 \in \{h_1, p_1, s_1, n_1\}$. Clearly from the above equation, l_0 and l_1 can not be determined. However, a similar equation can be obtained for another query. If the two queries is the same type and have the same storage structure, then the actual values of l_0 and l_1 can be obtained by solving the equations. And in turn, these values can be used to modify the parameters of the cost functions in the same way as the data transfer rate.

In addition, the size of the result of a query is also estimated. Usually this estimation is based on the assumption that attribute values are uniformly and independently distributed. This is known [5] to yield significant errors. If this is the case, the same adaptation procedure can be applied to adjust the size estimation formulas to reflect the reality [17].

6 Load Balancing for PRS

Our environment is based on the concept of shared-nothing [16] architecture in which processors do not share disk drives and random access memory. The sites over which a database is distributed are connected in a local area network (Ethernet). Each site is managed by a local DBMS. A given query is submitted to a front end system which parses the query and determines a strategy to process the query. The strategy usually consists of decomposing the query into several subqueries and assigning them to the local DBMSs at different sites. The site which receives the query is called the query site and all other sites involving in processing the query are called non-query sites. The partial results of the processing sites are sent to the query site and then combined to produce the final result. Relations may be either fragmented horizontally at different sites or unfragmented. Duplicate copies of the same relation/fragment are allowed to be placed at different sites.

6.1 PRS Algorithm

In this paper, we consider the situation where relations are unfragmented. In order to allow parallel processing, the PRS strategy [20] is used. Planning the strategy will include deciding which relation, if any, and which copy of the relation should be partitioned at what site. The PRS algorithm is briefly described as follows.

Assume the set of sites is S = S_1, S_2, ..., S_m. 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.

Now let us illustrate the PRS strategy with the help of an example.

Example 1 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 1. Assume the query is submitted to the system at S_1.

REL	#TUPLES	S_1	S_2	S_3
R_1	10000	R_1		
R_2	7000		R_2	R_2

Table 1: Distribution of relations for example 1

The PRS strategy first considers the strategy of processing the query at a single site without partitioning any relation. The response time using site S_1 as the processing site will be the sum of the time to transfer R_2 from S_2 or S_3 to S_1 and the time to perform the join at S_1. The response time using S_2 (S_3) as the processing site will be the sum of the time to transfer R_1 to S_2 (S_3), the time to perform the join at S_2 (S_3), and the time to send the result to S_1.

Now, we consider partitioning R_1 into two fragments. There are three approaches for choosing the processing sites: $\{S_1, S_2\}$, $\{S_2, S_3\}$, and $\{S_1, S_3\}$. The PRS algorithm estimates the response time for each approach. Let us take $\{S_2, S_3\}$ as an example. In this case, one fragment, say F_1, is sent to S_2 and the other fragment, say F_2, is sent 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 single site processing strategy since only a fragment is joined with R_2 instead of the whole relation. However, partitioning of R_1 takes time and the partial results have to be transferred to S_1 and then combined. To estimate the response time of the strategy, the PRS algorithm first estimates the total times at the individual sites (S_1 and S_2). For example, the total time at S_1 is the sum of the time to partition R_1 at S_1, the time to transfer F_1 to S_2, the time to perform the join ($F_1 \bowtie R_2$), and the time to send the result to S_1. Similarly, the total time at S_2 can be estimated. The sizes of the two fragments are chosen in such a way that the total times at S_2 and S_3 are approximately the same. Therefore, the response time of the query will be the sum of the total time and the time to combine the partial results.

Finally, we consider partitioning R_1 into three fragments. In this case, one fragment remains at S_1 and the other two fragments are sent to S_2 and S_3 respectively. Again, the response time is the sum of the maximum of the total times at the individual sites and the time to combine the partial results.

Similarly, the PRS algorithm considers partitioning R_2 into two/three fragments at site S_2 and partitioning R_2 into two/three fragments at site S_3.

Among all the strategies considered, the PRS algorithm picks the one which has the smallest response time. For this example, if S_3 is slightly faster than S_2 but much faster than S_1 and the size of the result is not too big, the strategy of partitioning R_1 into two fragments and processing the subqueries at S_2 and S_3 is likely to be faster than the single site processing strategies. ■

6.2 Execution Plan Refining

In Section 4, we have discussed how the costs of various type of queries can be estimated in a multi-user environment by incorporating the load information. Now let us discuss how to apply the techniques to the PRS algorithm so that load balancing can be achieved in a multi-user environment. As we have mentioned earlier, the input to this phase is a set of execution plans generated by the query optimization algorithms. Refining of the execution plans consists of the following two steps:

- For each of the plans, balance the load by moving some processing from the heavily loaded sites to the sites that have light load.
- Choose the execution plan that has the smallest response time in the multi-user environment from the refined set of plans.

Example 2 Let the query be $R_1 \bowtie R_2$ and the distribution of the relations be as shown in Table 2. Assume all the three sites have equal processing speeds and both relations do not have fast access paths.

Relation	Total tuples	S_1	S_2	S_3
R_1	15000	R_1		R_1
R_2	12000		R_2	

Table 2: Data distribution for example 2

During query optimization, the following plans could have been generated:

P_1: Partition R_1 at site S_1 into three fragment F_{11}, F_{12} and F_{13}, and send F_{12} to S_2 and F_{13} to S_3; replicate R_2 at S_1 and S_3; process a join ($F_{1i} \bowtie R_2$) at site i (i=1,2,3); and then combine the partial results.

P_2: Similar to P_1 but partition the copy of R_1 at site S_3.

P_3: Partition R_2 at S_2 into three fragments (F_{21}, F_{22} and F_{23}), send F_{21} and F_{23} to S_1 and S_3 respectively, and replicate R_1 at site S_2 by either send it from S_1 or send it from S_3; process a join at each site; and then combine the partial results.

If we use $T(P)$ to represent the response time for plan P, then in a single user environment we have $T(P_1) = T(P_2) < T(P_3)$. Thus, either P_1 or P_2 can be arbitrarily chosen as the query plan to be executed. Let us assume that PRS chooses P_1. If we use $T(P, S)$ to represent the total time spent at site S with plan P, then the following can be assumed:

$$T(P_i, S_1) = T(P_i, S_2) = T(P_i, S_3) \qquad (i = 1, 2, 3)$$

However, if multiple users exist, the above will not be true. For example, if S_1 is heavily loaded, then we will likely to have

$$T(P_i, S_1) > T(P_i, S_2) = T(P_i, S_3) \qquad (i = 1, 2, 3)$$

That is the total cost at S_1 will be higher than those of the other two sites. This problem can be solved by reassigning the tuples of the relation being partitioned such that load balancing can be achieved when

multiple users exist. In other words, relation R_i is partitioned into F_{i1}^m, F_{i2}^m and F_{i3}^m, where $F_{i1}^m < F_{i1}$, $F_{i2}^m > F_{i2}$ and $F_{i3}^m > F_{i3}$. Similarly, we can refine the other two plans. Clearly the modified plans will give better response times than the original plans. Note however that after the modification, the plan originally chosen by PRS may not be the best plan any more. Therefore, we have to choose the best plan among several modified plans by comparing their response times. If only S_1 is heavily loaded, P_2 will be superior to P_1. However, if both S_1 and S_3 are heavily loaded, P_3 may become the best plan. ■

7 Conclusion

Query processing is a very important issue in distributed databases. Many algorithms have been proposed to process distributed queries efficiently. However, our experience and experimental results with the PRS algorithm indicate that load balancing is not achieved because of oversimplified cost models and diversity of DBMSs in a multi-database system. In this paper, we provide an adaptive scheme to do load balancing effectively. The scheme takes into account a multi-user environment in which the load at different sites varies. The PRS algorithm is used to explain how to achieve load balancing in a multi-user environment. The scheme also has learning capability which allows dynamic cost estimation. The cost functions can be adaptively adjusted as the environment changes.

We plan to implement the learning scheme and integrate it with our distributed query processing system which was implemented on top of existing relational DBMSs[13] and do performance evaluation in a realistic environment.

References

[1] P. Agrawal, D. Bitton, K. Guh, C. Liu, and C. Yu. A case study for distributed query processing. *Proc. Int'l Symp. on Databases in Parallel and Distr. Sys.*, pp. 124–136, Austin, TX, Dec. 1988.

[2] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking Database Systems – A Systematic Approach. *Proc. of the 9th VLDB Conf.*, pp. 8–19, Florence, Italy, Oct. 1983.

[3] M. Carey, M. Livny, and H. Lu. Dynamic task allocation in a distributed database system programs. *Proc. 5th Int'l Conf. on Distr. Computing Sys.*, pp. 17–28, Denver, CO, May 1985.

[4] M. Carey and H. Lu. Load balancing in a locally distributed database system. In *Proc. 1986 ACM SIGMOD Conf.*, pp. 108–119, Washington DC, May 1986.

[5] S. Christodoulakis. Estimating block transfer and join sizes. In *Proc. 1983 ACM SIGMOD Conf.*, pp. 40–54, San Jose, CA, May 1983.

[6] S. Christodoulakis. Estimating record selectivities. *Information Systems*, 8(2):69–79, 1983.

[7] J. Fedorowicz. Database performance evaluation in an indexed file environment. *ACM TODS*, 12(1):85–110, March 1987.

[8] W.C. Hou, G. Ozsoyoglu, and B. Taneja. Statistical estimators for relational algebra expressions. In *Proc. ACM PODS*, pp. 276–287, New York, March 1988.

[9] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[10] R.J. Lipton and J.F. Naughton. Practical selectivity estimation through adaptive sampling. In *Proc. 1990 ACM SIGMOD Conf.*, pp. 1–11, Atlantic City, NJ, May 1990.

[11] C. Liu and I. Chu. Load balancing for distributed databases. In *Proc. of the 1994 Int'l Comp. Symp.*, Taiwan, Dec. 1994.

[12] C. Liu and C. Yu. Validation and performance evaluation of the partition and replicate algorithm. In *Proc. of the 12th IEEE Int'l Conf. on Distr. Computing Sys.*, pp. 400–407, Yokohama, Japan, June 1992.

[13] C. Liu and C. Yu. Performance issues in distributed query processing. *IEEE Trans. on Par. and Distr. Sys.*, 4(8):889–905, Aug. 1993.

[14] W. Meng, C. Liu, W. Sun, and C. Yu. Predict query processing cost in a distributed database system. In *Proc. of the 4th Int'l Conf. on Database and Expert Systems Applications*, pp. 122–133, Prague, Czech Republic, Sep. 1993.

[15] G. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proc. 8th VLDB Conf.*, pp. 257–262, Mexico City, Mexico, September 1982.

[16] M. Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9, No. 1:4 – 9, 1986.

[17] C. Yu, L. Lilien, K. Guh, M. Templeton, D. Brill, and A. Chen. Adaptive techniques for distributed query optimization. In *Proc. 2nd IEEE Int'l Conf. on Data Eng.*, pp. 86–93, Los Angeles, CA, Feb. 1986.

[18] C. Yu and C. Liu. Experiences with distributed query processing. In *Proc. 6th IEEE Int'l Conf. on Data Eng.*, pp. 192–199, Los Angeles, CA, February 1990.

[19] C.T. Yu, K. C. Guh, W. Zhang, M. Templeton, D. Brill, and A.L.P. Chen. Algorithms to process distributed queries in fast local networks. *IEEE Transactions on Computers*, 36(10):1153–1164, Oct. 1987.

[20] C.T. Yu, K.C. Guh, D. Brill, and A.L.P. Chen. Partition strategy for distributed query processing in fast local networks. *IEEE TSE*, 15(6):780–793, June 1989.