DYNAMIC TASK ALLOCATION IN A DISTRIBUTED DATABASE SYSTEM

by

Michael J. Carey
Miron Livny
Hongjun Lu

Computer Sciences Technical Report #556

September 1984

Dynamic Task Allocation in a Distributed Database System

Michael J. Carey
Miron Livny
Hongjun Lu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was supported by the Wisconsin Alumni Research Foundation, National Science Foundation Grant Number DCR-8402818, and an IBM Faculty Development Award.

Dynamic Task Allocation in a Distributed Database System

Michael J. Carey
Miron Livny
Hongjun Lu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

This paper addresses the problem of dynamically assigning queries to sites in a distributed database system. This work, loosely related to work on load balancing in distributed operating systems, is unique in several ways. First, the system workload is comprised of several classes of tasks, each with different processing requirements. Second, estimates of the processing requirements of each task are available as a result of query optimization. Finally, "load" is a two-dimensional quantity in a distributed database system, as both the CPU and I/O loads of sites must be considered.

The dynamic query allocation problem is described, and the results of a study of optimal allocations for queries in a two-class, four-site environment are presented. This study leads to the identification of performance criteria that are appropriate for use in a multi-class environment. Several heuristic algorithms are given for achieving near-optimal query allocations dynamically. A simulation model of a fully-replicated distributed database system is used to study these heuristic algorithms and to compare them to an algorithm that simply balances the number of jobs at each site. The results of this study indicate that knowledge about task processing requirements can be used effectively to improve overall system performance by dynamically allocating queries to sites.

1. INTRODUCTION

1.1. The Problem

In distributed relational database systems, queries are often decomposed into sequences of *data moves* and *subqueries* [Aper83, Bern81, Epst78, Hevn79, Roth77, Sacc82, Seli80, Yu83]. Each subquery is a set of database operations that can be performed together at a single site. If some single site contains all of the data needed to execute an entire query, the query itself is the only subquery and no data moves are necessary. If no one site holds all of the data required by the query, one or more data moves are required to transfer subsets of the database from their storage sites to the selected execution sites for subqueries of the query. If the distributed database includes replicated data, data stored at several sites, it may be possible to execute a

This research was supported by the Wisconsin Alumni Research Foundation, National Science Foundation Grant Number DCR-8402818, and an IBM Faculty Development Award

given subquery at one of several sites containing the data which it references. In this case, it is necessary to somehow select a site at which to execute the subquery. We call this problem the *query allocation problem*.

Some of the previous papers on distributed query processing have ignored the problem of choosing from among several copies of the data of interest by assuming either that data is not replicated or else that one copy of each piece of data needed by the query, called a *materialization* of the database, has already been chosen [Aper83, Bern81, Epst78, Hevn79, Sacc82]. These papers then address the problem of selecting a good query execution plan given this materialization. Other papers have suggested ways to select from among several copies of a replicated relation by picking the one for which the estimated cost of executing the query is minimized [Seli79] or for which a heuristic estimator for this cost is minimized [Yu83]. These solutions are based upon the assumption that the database is partitioned in a way such that a 'best' copy of each relation R exists, i.e., that one copy of R has an access path that other copies do not have, or that one copy is stored at a site that has some data needed to execute the query that is not available at other sites that hold copies of R. If data partitioning is uniform, there may not be a best copy — all copies may be equally good.

Nearly all previous work on distributed query processing has focused on *static* allocation strategies. Processing sites are selected without regard for the amount of other work that the sites have to do, i.e., ignoring their dynamic load status. By ignoring the instantaneous load distribution of the system, a query might be allocated to a heavily loaded site while other sites that are capable of serving it have a light load or are even idle [Livn82, Livn83, McCo81]. A static allocation strategy may lead to unnecessary resource contention and thus to poor performance. As an extreme example, if everyone were to submit the same query to the system, it is possible (even likely) in most of the previous solutions that the same execution plan will be selected for each query, and only the few sites chosen for this plan will be busy. Truly 'optimal' plans for query execution must consider the *dynamic* system state as well as the static costs of query execution. Alonso has studied the relative performance of alternative query processing plans under various load distributions [Alon84]. His work has shown that a plan which is optimal under one load distribution is not necessarily optimal for other load distributions.

1.2. Dynamic Query Allocation

In this paper we present several *dynamic query allocation* strategies. The particular problem that we address here is that of allocating read-only queries to sites in a fully replicated distributed database system[†]. However, our results will also provide insight into how to go about allocating subqueries to one of a number of candidate sites in a system with partially replicated data.

We should point out that other opportunities for making load-based query processing decisions (besides selecting from among candidate copies of replicated data) do exist in distributed database systems. For example, several options exist for joining a pair of relations A and B that reside at different sites of the system — two options are to send A to B's site and perform the join there or to send B to A's site and perform the join there. This decision is usually made (statically) so as to minimize a cost measure such as the amount of data shipped over the communications subnet or the sum of the communications and local processing costs [Aper83. Bern81. Epst78. Hevn79, Roth77. Sacc82. Seli80. Yu83], but the dynamic state of the system could make the static decision sub-optimal in some cases [Alon84]. We leave the extension of our ideas on dynamic query allocation to this type of query allocation problem for later work.

Our work is loosely related to some of the previous work on load balancing in the operating systems area [Brya81, Livn83, Ni81, Ni82], but the nature of our environment leads to several important differences. These differences lie in our model of a site, what is known about tasks in our environment, the composition of the workload, and the times at which a task may reasonably be transferred from one site to another.

1.2.1. Multi-Server Sites

Previous work on load balancing has been concerned with systems that consist of only one type of resource (the CPU). Sites have been modeled as single server queuing systems with either First-Come-First-Served (FCFS) or Processor Sharing (PS) service disciplines. Queries require *two* types of resources, CPU and I/O resources, so a single server model is not appropriate for describing the behavior of a sites in a distributed database system. Thus, our work addresses the problem of load balancing in a system in which each site consists of these two types of resources.

[†]The read-only assumption is not a major problem, as updates must be propagated to all sites regardless of the processing site.

1.2.2. Knowledge of Processing Demands

Query optimizers put a good deal of effort into computing good estimates of the costs of executing queries in order to select the best execution plan [Seli79, DBE82]. We assume that estimates of the CPU and I/O needs of queries are 'attached' to the queries and available for use when selecting appropriate processing sites. Most other work on load-balancing has assumed that such knowledge is not available.

1.2.3. Multiple Query Classes

Another salient aspect of our work is that we assume that the workload is comprised of multiple classes of tasks. Queries can be CPU- or I/O-bound, and the amount of data that they reference can vary widely depending upon both the nature of the requests and the access paths (index structures) available for retrieving the desired data. Thus, in a distributed database environment, it is not appropriate to consider only workloads in which the characteristics of tasks are all drawn from a single distribution. Our study considers workloads comprised of two classes of tasks, CPU- and I/O-bound tasks, and each class has its own set of task characteristics. (More will be said about these characteristics in Section 2.)

1.2.4. When to Transfer a Task

A final difference between our work and previous load balancing work is that, at least for now, we do not consider moving a query once it begins executing at a site. Load balancing algorithms in operating systems, on the other hand, will usually consider moving tasks in any stage of execution, requiring a carefully designed and programmed task migration mechanism. Such a general mechanism must deal with forwarding messages, handling open I/O connections, and moving task state information (contents of registers, a task control block, program and data segments, etc.) [Powe83]. By assigning queries to sites when they begin running, we avoid the need for much of this complexity. Also, having a query access data remotely would be too costly, and it is unreasonable to have a query stop what it's doing and pick up again on a different copy of its data (due to active file scans and partially-written temporary files).

1.3. Outline

The remainder of this paper presents our results to date on developing and evaluating dynamic query allocation algorithms. Section 2 describes the distributed database system model that is used in the remainder of the paper. Section 3 describes a preliminary study of optimal allocations for queries in an environment with four sites and two classes of queries. This section also addresses the problem of evaluating query allocation criteria in a multi-class environment. In section 4 we present several heuristic algorithms for dynamic query allocation, and Section 5 describes a simulation study of these heuristics. The study compares our heuristic algorithms both with each other and with an algorithm that simply attempts to keep the number of tasks at each site balanced. Finally, Section 6 presents our conclusions and outlines our plans for future work.

2. MODELING A DISTRIBUTED DATABASE SYSTEM

Figure 1 depicts our model of a distributed database system. The system consists of a collection of database processing sites, or DB sites, each of which stores a copy of the database. The DB sites are connected together by a communications subnetwork. (We will focus our attention on local networks in this paper.) Each site has a collection of terminals from which queries originate and to which the results are

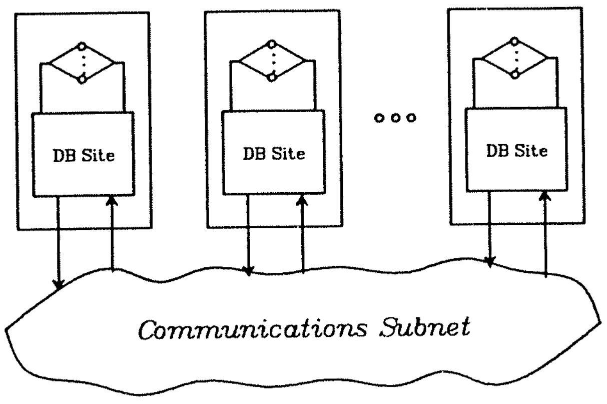

Figure 1: Distributed Database System Model.

returned, so our model is a closed queuing model. We assume throughout this paper that the system is completely homogeneous, i.e., that all sites are configured identically and have the same workload characteristics.

Figure 2 shows our model of a DB site. Each site has a set of terminals, several disks, a CPU, and an outgoing message queue. The diamonds in the figure depict decision points in the model. The decisions that are made by the dynamic query allocation software are represented by the doubly-outlined diamonds. When a query is initiated by a terminal, the query allocator looks at its processing requirements and decides whether to process the query locally or to transfer it to another site. The query starts its execution at the disk service center at its execution site. The query will be routed to this center either directly or via the communications

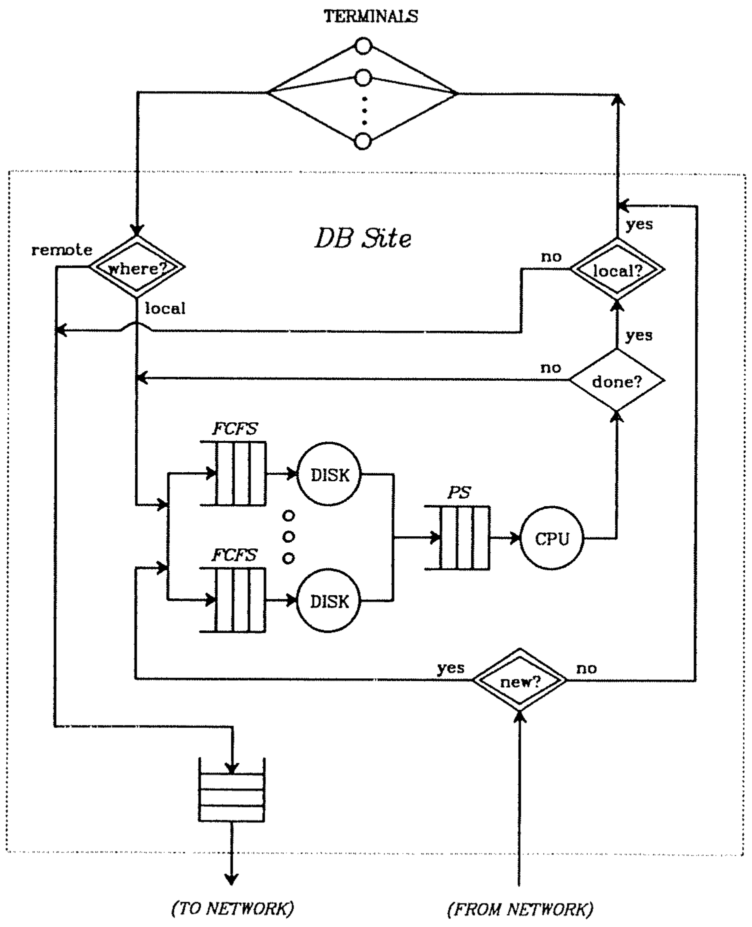

Figure 2: DB Site Model.

subnet depending on whether it is to be executed locally or remotely. Once the query has begun execution, it cycles through the disk and CPU service centers a number of times, reading and processing pages from the database.[†] The disks are modeled as FCFS servers, and the CPU is modeled as a PS server. Once finished, the query is returned to its terminal of origin. In the case of a remotely executed query, this requires sending the results of the query back to its home site.

Table 1 summarizes the parameters associated with each of the DB sites. The storage hardware at each DB site is described by two parameters. The parameter *num-disks* is the number of disks at the site, and *disk-time* is the mean time required to access a disk page. The workload at each site is characterized by three parameters — *mpl*, the number of terminals or multiprogramming level for the site; *class_prob*, the class distribution function that determines the probability that a newly generated query will be a class C query; and *think_time*, the mean think time for the terminals.

We have decided to consider only two classes of queries in this study. Each query class Q_j is characterized by four parameters, $page\_cpu\_time_j$, num_reads_j, $result\_fraction_j$, and $query\_size_j$. Table 2 summarizes these class parameters. The *page_cpu_time* parameter is the mean CPU time required to process

DB Site Parameters	
num_disks	number of disks per site
disk_time	mean access time for a disk page
mpl	number of terminals per site
think_time	mean terminal think time
class_prob	class distribution function

Table 1: DB site parameters.

Class Parameters	
page_cpu_time	mean per-page CPU demand for the class
num_reads	mean number of reads for the class
result_fraction	mean fractional result size for the class
query_size	query descriptor size (in bytes) for the class

Table 2: Class parameters.

[†]We assume that the results of each query are accumulated in main memory as the query executes

one page of data read from the disk for queries of the class. The parameter *num_reads* is the mean number of times that queries of the class cycle through the I/O and CPU service centers, i.e., the mean number of disk pages that they read. The parameter *result_fraction* is the mean number of result pages produced for queries of the class (expressed as a fraction of the total number of pages read). Finally, the parameter *query_size* is the number of bytes required to describe a query of the class (i.e., the amount of data that would have to be transferred to initiate the query remotely).

Our model of the communications subnetwork is a simple token-ring style local network model. The network has a single message buffer for each site, and sites are polled in a round-robin fashion for requests to send messages. The cost of sending a message is a linear function of the length of the message. When the network finds a site that is ready to send a message, it sends its message, delays for the appropriate amount of time, and then continues on with the polling process. We assume that the overhead of the polling process is negligible in this study. The parameters related to the communications costs are listed in Table 3. The parameter *msg_time* is the time required to send one byte of data over the network. The parameter *page_size* is the size of a disk page in bytes (which plays a role in determining the message size for query results). One last assumption that should be mentioned is that we neglect the overhead associated with load status messages, assuming for the purpose of this study that each site knows the current loads of all other sites in the system. (We will discuss this assumption in more detail in a later section.)

3. OPTIMAL QUERY ALLOCATION

As described earlier, the execution of a query requires two types of resources, the CPU and I/O resources. A query is a sequence of I/O and CPU requests that are generated one after another. Interleaving the execution of a number of queries will lead to concurrent activity on both types of resources, in turn

Parameters Related to Communications Costs	
msg_time	network transfer time for one byte of data
page_size	size of disk page in bytes

Table 3: Parameters related to communications costs.

reducing system response time. It is this fact that motivated the introduction of multiprogramming systems. Shorter waiting time, however, is not the only reasonable goal for a scheduling mechanism for a timesharing environment. Another important goal is to somehow provide an environment that will give users 'fair' response times by some definition.

Most users would agree that the expected waiting time (i.e., queuing time) of a task with a given execution time, $\overline{W}(x)$, can serve as a measure of fairness for a scheduling policy. Two users with service demands x_1 and x_2 will consider the scheduling policy as being fair if the ratio of their respective expected waiting times is equal to x_1/x_2. Another way of saying this is that if the normalized mean waiting time of a user, $\hat{W}(x)$, is defined as the ratio of $\overline{W}(x)$ to x, then all users would like to have the same $\hat{W}(x)$ regardless of the number of units that are needed to fulfill their tasks. Thus, the difference between the normalized mean waiting times of two tasks gives a quantitative indication of the degree to which a system is biased towards a particular type of service demand.

In order to reduce variation in the normalized mean waiting times for different users, round-robin (RR) scheduling schemes are often used for resources that can support a preemptive resume scheme with a minimal amount of overhead. If the time quanta of the RR algorithm shrinks to zero, which leads to processor sharing (PS), then all tasks have the same normalized response time on a $M/G/1$ system [Klei75]. However, this property of the PS scheduling strategy does not necessarily hold when a number of PS servers are combined into one integrated system. A difference in the utilization of the system resources will lead to discrimination against those users that depend more heavily on the highly utilized resources for service. Thus, biased response time is not necessarily due to a biased allocation strategy.

As described in Section 1, the goal of this paper is to investigate the extent to which the load balancing process can benefit from estimates of query resource demands that a query optimizer can provide. In order to get a quantitative indication of this, we have studied the effect of a single allocation decision on the performance of a group of the DB sites in the model of Section 2 under a static load distribution. A four-site system with two classes of queries is assumed. The values that have been assigned to the relevant model parameters are summarized in Table 4. Note that both the number of cycles and the thinking time are assumed to

System Parameters		
num_sites	4	
DB Site Parameters		
num_disks	2	
disk_time	1	
think_time	large	
Class Parameters	I/O Bound	CPU Bound
page_cpu_time	0.05 - 0.5	1.00 - 0.25
num_reads	large	large
msg_time	0	0

Table 4: Parameter settings for optimal allocation study.

be large since only static (or steady-state) load distributions are addressed here. Message time is also ignored for the purpose of this part of the study. All service times used in this section are assumed to have an exponential distribution.

The load distribution of the system is defined by the matrix $L=[l_{i,j}]$, where $l_{i,j}$ is the number of class i queries being served by site j. We denote an arrival of a class i query at a system with a load distribution L by $A(L,i)$. The Mean Value algorithm [Reis78] has been used for deriving the various performance measures in this study. The values obtained reflect the performance of the system during a time period beginning after the new query has begun service at the selected site and ending upon the first departure or arrival of a query; we assume that this period is long enough for steady-state behavior to be achieved.

When no information is available about query service demands, the allocation decision can rely only on the instantaneous query distribution of the system as given by the vector $N=[n_1, \ldots, n_4]$ where $n_i=l_{1,i}+l_{2,i}$. Under these circumstances, the goal of the allocation policy will most likely be to minimize the query-difference (QD) of the system, defined as the maximum of $|n_i-n_j|$ for $i,j=1,\ldots,4$. Such a strategy, where the goal is simply to *balance the number of queries* at each site, is clearly not an optimal one. In a multi-class environment, the expected waiting time of a query can be reduced if the allocation mechanism relies on the instantaneous load distribution of the system. For a given arrival, $A(L,i)$, a quantitative measure of how much better can a query allocation do if it can use the instantaneous load distribution of the system for its decisions instead of just the current query distribution is given by the Waiting Improvement

Factor:

$$WIF(L,i) = \frac{\overline{W}_{BNQ}(L,i) - \overline{W}_{OPT}(L,i)}{\overline{W}_{BNQ}(L,i)}$$

$\overline{W}_{BNQ}(L,i)$ in this expression is the expected waiting time per cycle when the 'minimal QD' strategy is applied for $A(L,i)$, and $\overline{W}_{OPT}(L,i)$ is the minimum of the expected waiting time per cycle over all possible allocations of $A(L,i)$ (i.e., the expected waiting time for the optimal allocation).

Table 5 presents $WIF(L,i)$ for different arrival conditions and class mixtures. L in the table is the load distribution matrix prior to the new query's arrival (i.e., the number of queries of each class at each of the four sites); i is the class of the newly arriving query. One parameter varied is the ratio of the mean CPU demands of the two classes. In most of the cases that were studied, the improvement in the expected waiting time of a query exceeds 10%. For some arrivals, waiting time can be reduced by more than 30% if the class of the queries is known. Note that in the case of the first four class mixtures, an increase in the ratio of the mean CPU demands of the classes produces an increase in the Waiting Improvement Factor. However, there are cases (the two last mixtures) where an increase in the ratio causes a decrease in WIF. Another parameter varied in Table 5 is the total number of queries in the system (which increases from left to right in the table). From the results obtained we can conclude that an increase in the number of queries, and thus an increase in system utilization, decreases the beneficial impact that resource demand estimates may have on the allocation process.

L	1 1 0 0 0 0 1 1		1 1 1 0 0 0 0 1		2 1 0 0 0 0 1 1		2 1 1 0 0 0 0 1		2 1 2 0 0 0 0 1		2 1 1 0 0 1 1 2	
i	1	2	1	2	1	2	1	2	1	2	1	2
cpu_1/cpu_2												
.05/0.5	.14	.01	.08	.01	.05	.01	.10	.01	.01	.09	.05	.05
.05/1.0	.24	.13	.14	.18	.09	.07	.16	.04	.09	.04	.11	.04
.10/1.0	.20	.12	.11	.16	.07	.06	.13	.03	.08	.03	.09	.03
.10/2.0	.31	.31	.19	.41	.18	.11	.20	.10	.11	.09	.09	.15
.50/2.0	.00	.22	.00	.30	.00	.16	.01	.09	.01	.09	.05	.05
.50/2.5	.02	.17	.01	.23	.01	.11	.01	.06	.01	.06	.03	.04

Table 5: Waiting Improvement Factor $WIF(L,i)$.

Table 5 clearly shows that the use of additional information can have an advantageous impact on the expected waiting time of the system. However, since fairness is also an important aspect of the scheduling process, and because the desires for fair response time and for minimal waiting time may conflict, we have analyzed the extent to which different allocation decisions discriminate against one of the query classes. We use the difference in the normalized expected waiting time of the two classes as a measure of the fairness of a given allocation. We have defined a Fairness Improvement Factor of an arrival $A(L,i)$ in a manner analogous to the Waiting Improvement Factor. We define this measure of fairness improvement to be:

$$FIF(L,i)=\frac{F_{BNQ}(L,i)-F_{OPT}(L,i)}{F_{BNQ}(L,i)}$$

$F_{BNQ}(L,i)$ in this expression is the absolute value of the difference between the normalized expected waiting time of the two classes when the 'minimal QD' strategy is applied for $A(L,i)$, and $F_{OPT}(L,i)$ is the minimum of the absolute value of this difference for all possible allocations of $A(L,i)$. The values of $FIF(L,i)$ for the same arrivals that were used in the analysis of $WIF(L,i)$ are summarized in Table 6. The results presented in the table show that in all cases a significant improvement in the fairness of the system can be achieved. However, no clear relationship between the arrival conditions and values of $FIF(L,i)$ is apparent from the table.

This study of the properties of a single allocation was motivated by a desire to explore the potential of using resource demand estimators. The results obtained indicate that information on the load distribution of

L	1 1 0 0 0 0 1 1		1 1 1 0 0 0 0 1		2 1 0 0 0 0 1 1		2 1 1 0 0 0 0 1		2 1 2 0 0 0 0 1		2 1 1 0 0 1 1 2	
i	1	2	1	2	1	2	1	2	1	2	1	2
cpu_1 / cpu_2												
.05/.50	.69	.60	.64	.11	.42	.48	.69	.20	.89	.79	.72	.87
.05/1.0	.75	.70	.70	.01	.38	.60	.89	.07	.70	.93	.68	.67
.10/1.0	.72	.69	.67	.02	.39	.72	.79	.05	.77	.74	.52	.55
.10/2.0	.78	.81	.73	.30	.36	.60	.99	.22	.60	.25	.48	.69
.50/2.0	.34	.95	.88	.35	.75	.14	.11	.83	.40	.55	.84	.77
.50/2.5	.60	.74	.56	.07	.50	.15	.40	.55	.75	.25	.47	.95

Table 6: Fairness Improvement Factor $FIF(L,i)$.

the system can significantly improve both the performance and the fairness of the allocation process. However, minimal expected waiting time and maximal fairness may not be achieved together, and thus a decision has to be made about which goal is more important. $W_{OPT}(L,i)$ and $F_{OPT}(L,i)$ were achieved by different allocations in about half of the cases that were examined in this study. In the next section we will present three allocation strategies that aspire to minimize the expected waiting time of queries, and in Section 5 we will evaluate their impact on the performance of the system.

4. HEURISTICS FOR DYNAMIC QUERY ALLOCATION

The previous section has shown that knowledge about the CPU and I/O demands of tasks can, in theory, be used to reduce waiting time and thus improve response time for queries. In this section we present two heuristic algorithms for choosing the processing site for an arriving task — the goal of each is to dynamically achieve good query allocations. Before we describe these heuristics, though, we describe an allocation algorithm that simply tries to balance the number of tasks at the sites (i.e., that tries to minimize the query difference QD defined in the previous section). This balancing goal is similar to the approach that several previous load balancing algorithms have used [Livn82, Livn83, Ni81, Ni82], and it will provide a non-information-based query allocation algorithm with which to compare the performance of our heuristics later on. All of the allocation algorithms presented here can be viewed as choosing the processing site with the minimum estimated processing cost, where the cost estimation procedure varies from algorithm to algorithm. Figure 3 describes the selection process in this manner. (One detail not shown in the figure is that the 'foreach' loop that examines possible remote execution sites should scan these sites in a round-robin fashion.)

4.1. Balance the Number of Queries

As mentioned above, one simple heuristic for dynamic query allocation is to try and keep the *number* of queries at each site evenly balanced. We call this approach the BNQ ('Balance the Number of Queries') algorithm. Each site knows the current query distribution of the system. Then, when a new query is initiated from a terminal at some site, that site routes the query to the site with the smallest number of queries for processing. Figure 4 gives the site cost function for the BNQ heuristic.

```
function SelectSite(q: query; arrival_site: site): site;
var
   cur_cost, min_cost: real;
   remote_site, best_site: site;
begin
   best_site := arrival_site;
   min_cost := SiteCost(q, arrival_site);
   foreach remote_site in {sites} - arrival_site do begin
      cur_cost := SiteCost(q, remote_site);
      if cur_cost < min_cost then begin
         min_cost := cur_cost;
         best_site := remote_site;
      end;
   end;
   SelectSite := best_site;
end;
```

Figure 3: Function to select processing site for query.

```
function SiteCost(q: query; s: site): integer;
begin
   SiteCost := Num_Queries(s);
end;
```

Figure 4: Cost function for BNQ algorithm.

4.2. Balance the Number of Queries by Resource Demands

Our first information-based heuristic is a simple extension of the BNQ algorithm. This heuristic, called the BNQRD ('Balance the Number of Queries by Resource Demands') algorithm, requires that each site knows the number of CPU-bound queries and the number of I/O-bound queries at every site in the system. When a new query is initiated from a terminal at some site, this query is classified as being either an I/O-bound or a CPU-bound query based on the knowledge of its resource demands. The query is then routed to the site with the smallest number of queries of the same type (i.e., I/O-bound or CPU-bound).

To classify a query as being I/O- or CPU-bound when each site has just one disk, the I/O and CPU demands of the query (based on the output of the query optimizer) are compared. If the I/O demand is greater, it is an I/O-bound query; otherwise it is CPU-bound. In the case of multiple disks per site, the I/O demand per disk, defined as the disk access time divided by the number of disks per site, is first computed.

This per-disk I/O demand is then compared with the CPU demand of the query, as before, to classify it as an I/O- or CPU-bound query. Figure 5 gives the site cost function for BNQRD.

4.3. Least Estimated Response Time

Our second information-based heuristic, LERT (for 'Least Estimated Response Time'), uses the I/O and CPU demand information for a newly arrived query to estimate the response time of the query at each site in the system and then to route the query to the site with the least estimated response time. This approach is related to the work of Bryant and Finkel [Brya81]. As in the BNQRD algorithm, each site must know the number of CPU- and I/O-bound queries at all sites in the system.

```
function SiteCost(q: query; s: site): integer;
begin
   if (disk_time / num_disks) > Page_CPU_Time(q) then begin
      SiteCost := Num_IO_Queries(s);
   end else begin
      SiteCost := Num_CPU_Queries(s);
   end;
end;
```

Figure 5: Cost function for BNQRD algorithm.

```
function SiteCost(q: query; s: site): real;
var
   cpu_time, io_time, net_time: real;
   cpu_wait, io_wait: real;
begin
   cpu_time := Num_Reads(q) * Page_CPU_Time(q);
   io_time := Num_Reads(q) * disk_time;
   if s = arrival_site then begin
      net_time := 0.0;
   end else begin
      net_time := Transfer_Time(q) + Return_Time(q);
   end;
   cpu_wait :=  cpu_time * Num_CPU_Queries(s);
   io_wait :=  io_time * (Num_IO_Queries(s) / num_disks);
   SiteCost := cpu_time + cpu_wait + io_time + io_wait + net_time;
end;
```

Figure 6: Cost function for LERT algorithm.

The response time computation itself, given as a cost function in Figure 6, is based on several simplifying approximations: First, it is assumed that competition for a given resource type at a site is only with those queries at the site that rely more heavily on this type of resource. (That is, a query will compete only with I/O-bound queries for I/O service at a site, and it will compete only with CPU-bound queries for CPU service at the site.) Second, it is assumed that both the CPU and the disks at each site have PS service disciplines. This is accurate for the CPU, but it is only a rough approximation for the disks. The disks actually have more of a round-robin flavor — requests for page accesses are served on an FCFS basis, and each query cycles through the disk queue a number of times making page access requests. Third, it is assumed (for lack of better information) that the number of CPU- and I/O-bound jobs at each site will not change during the execution of the newly-arrived query. Thus, the cost of executing a query at a site is the sum of its service demands at the site, the message costs for sending the query to the site and returning its results to the site of origin (if the execution site is not the site of origin), and the waiting time for CPU and I/O service based on the number of competing jobs at the site.

4.4. Discussion

The three algorithms described in this section represent three different approaches to dynamic query allocation. The BNQ approach is based only on the query distribution, whereas the BNQRD and LERT approaches are based on more load distribution information. Thus, only the latter two approaches use information about the resource demands of queries. The reader has probably noticed that we have not yet said anything about *how* site state information is to be exchanged among the sites — this is intentional. Our attention in this study is focused solely on the issue of how such information can be used and the extent to which performance can be improved through its use. A good information exchange policy will not overburden either the sites or the communications subnetwork, and yet it will provide the sites with information that is sufficiently current so that performance improvements offered by the heuristics are not lost; we leave the design and analysis of such a policy for future work.

5. A SIMULATION STUDY

We saw in Section 3 that using information about loads and the resource demands of queries can potentially lead to improved performance. Since the BNQRD and LERT algorithms use more information than the BNQ algorithm, we expect BNQRD and LERT to outperform the BNQ algorithm. Also, only the LERT algorithm takes the cost of sending a query to a remote site, i.e., the message costs for sending the query elsewhere returning its results to the site of origin, into account. We expect that the inclusion of these costs will make the LERT algorithm less likely to call for unprofitable query transfers than the other algorithms. Thus, LERT should outperform BNQRD, and BNQRD should outperform BNQ. We will investigate the degree to which this is true in this section, where we report the results of a study of the performance of the three dynamic query allocation algorithms.

5.1. Simulation Details

In order to evaluate our algorithms, we have developed a simulation model of a distributed database system and its workload. Our model for a fully-replicated distributed database system is an implementation of the model described in Section 2 of this paper. This model has been implemented in the DISS simulation language [Melm84] and runs on an IBM 4341.

System Parameters		
num_sites	2 - 10	
DB Site Parameters		
num_disks	2	
disk_time	1	
disk_time_dev	20%	
mpl	15 - 30	
think_time	150 - 450	
class$_{io}$*_prob*	0.3 - 0.8	
Class Parameters	I/O Bound	CPU Bound
page_cpu_time	0.01 - 0.4	1.04 - 0.65
num_reads	20	20
msg_length	1	1

Table 7: Parameter settings for the simulations.

In the experiments reported here, the model parameters from Section 2 are set to the values listed in Table 7. The probability that a query is an I/O-bound query is $class_{io}\_prob$; the other class of query is CPU-bound. The number of reads for queries has an exponential distribution with a mean of *num_reads*. Disk service times (to access a page of data) are uniformly distributed on the range $disk\_time \pm disk\_time\_dev$. CPU service times have an exponential distribution with *page_cpu_time* as the mean. Think times at the terminals are exponentially distributed with mean *think_time*. The model parameters *result_fraction*, *query_size* and *msg_time* are currently combined into a single parameter, *msg_length*, which is a constant value representing the number of time units needed to send (or return) a query to (or from) a remote site. For the parameters which are shown in Table 7 as varying over some range, their values when not being varied are as follows: $num\_sites = 6$, $mpl = 20$, $class_{io}\_prob = 0.5$, $think\_time = 350$, and $page\_cpu\_mean = 0.05$ and 1.0 (for the two query classes).

5.2. Simulation Results

As discussed in section 3, the performance metrics of interest in this paper are the mean waiting time of a query, $\overline{W}$, and the diference in normalized waiting times between query classes 1 and 2, the fairness measure F. We have performed several experiments to investigate how our dynamic query allocation algorithms affect performance in terms of these metrics.

We first investigated the effects of the different algorithms on $\overline{W}$ under different system load situations. One way to obtain different system loads is to vary the *think_time* of queries. Table 8 presents the results of

think_time	ρ_c	$\overline{W}_{LOCAL}$	$\Delta \overline{W}_{X,LOCAL} / \overline{W}_{LOCAL}$ (%)			$\Delta \overline{W}_{X,BNQ} / \overline{W}_{BNQ}$ (%)	
			BNQ	BNQRD	LERT	BNQRD	LERT
150	0.85	72.71	4.89	17.03	14.84	12.76	10.46
200	0.77	48.61	10.30	23.08	24.61	14.25	15.96
250	0.68	35.71	23.55	32.30	32.67	11.44	11.92
300	0.59	26.82	26.54	38.43	37.43	16.19	14.82
350	0.53	22.71	38.53	41.96	43.54	5.57	9.58
400	0.48	18.37	38.02	40.84	42.72	4.55	7.58
450	0.43	15.60	41.13	44.27	46.50	5.33	9.12

Table 8: Waiting time versus think time.

such an experiment. $\overline{W}_{LOCAL}$ is the mean waiting time when queries are always processed locally (i.e., at their arrival site). The performance of the dynamic query allocation algorithms is expressed in terms of percentage improvements with respect to both the local case and to the BNQ algorithm. $\Delta \overline{W}_{X,LOCAL} / \overline{W}_{LOCAL}$, where X = {BNQ, BNQRD, LERT}, is the improvement obtained via dynamic allocation for each of the algorithms. The improvement obtained by using load information instead of simply using the query distribution is $\Delta \overline{W}_{X,BNQ} / \overline{W}_{BNQ}$, where X = {BNQRD, LERT}. We can see from the table that each dynamic allocation algorithm leads to a significant decrease in $\overline{W}$ with respect to local processing. More improvement is achieved when system utilization is lower since then there is a better chance of being able to allocate a newly arrived query to a lightly loaded or perhaps even idle DB site. Also, as we expected, BNQRD and LERT outperform BNQ by using the information about query resource demands and the distribution of the query classes at each site. LERT performs a bit better than BNQRD in most cases, but the difference is not tremendous. A possible explanation for this is that the message transfer time is small (1.0) as compared with execution time (30.5) in this case. When we tried changing *msg_length* to 2.0, the values of $\Delta \overline{W}_{X,BNQ} / \overline{W}_{BNQ}$ changed to 16.43 and 24.12 for X = BNQRD and LERT, respectively, when *think_time* = 350. That is, a larger diference was observed with a larger message time. This is because LERT considers this time when selecting a site, but BNQRD does not.

Various system loads can also be obtained by varying the number of terminals (*mpl*) at each DB site. Table 9 shows the effects of *mpl* on $\overline{W}$ for the different allocation algorithms. The trends observed in this experiment are similar to those observed previously. From a multiprogramming level viewpoint, the effect of

mpl	ρ_c	$\overline{W}_{LOCAL}$	$\Delta \overline{W}_{X,LOCAL} / \overline{W}_{LOCAL}$ (%)			$\Delta \overline{W}_{X,BNQ} / \overline{W}_{BNQ}$ (%)	
			BNQ	BNQRD	LERT	BNQRD	LERT
15	0.41	13.81	36.86	44.20	43.10	11.63	9.88
20	0.53	22.71	38.53	41.96	43.54	5.57	9.58
25	0.65	33.90	30.68	36.55	37.15	8.46	9.33
30	0.75	50.97	23.12	33.83	34.56	13.96	14.88
35	0.83	73.72	10.97	24.21	26.32	14.87	17.24

Table 9: Waiting time versus *mpl*.

Expected Response Time	Maximum *mpl*	
	LOCAL	LERT
≤ 40.0	10	17
≤ 50.0	18	23
≤ 60.0	21	28
≤ 70.0	27	31
≤ 80.0	29	34

Table 10: Maximum *mpl* versus response time.

dynamic query allocation is that the multiprogramming level of each of the DB sites can be increased without decreasing the mean query response time. In other words, the capacity of the system can be increased through dynamic query allocation. This point is illustrated by Table 10, which shows the maximum number of terminals for which the given expected response time can be provided using two different allocation algorithms (local processing versus LERT). Using LERT, we can increase the number of terminals at each site by 20-50 percent while providing the same response time as the system with fewer terminals when queries are simply processed locally.

Table 11 is the result of another experiment which examines the effect of the number of DB sites in the system (*num_sites*) on $\overline{W}$. The waiting time reduction due to dynamic allocation reaches a maximum and then decreases as *num_sites* is varied from 2 to 10. This is expected since increasing the number of DB sites in a distributed database system has two competing effects: On one hand, it improves the probability that a query will be allocated to an idle or a lightly loaded site. On the other hand, however, it increases competition for the commmunication channel and thus increases the waiting time for messages. To illustrate this

num_sites	$\overline{W}_{LOCAL}$	$\Delta \overline{W}_{X,LOCAL} / \overline{W}_{LOCAL}$ (%)		Subnet Utilization (%)		
		BNQ	LERT	Local	BNQ	LERT
2		15.19	26.82		6.35	6.49
4		27.10	33.54		21.38	20.90
6	21.53	34.18	39.18	0.0	37.07	36.04
8		32.17	39.23		54.41	52.07
10		26.13	36.27		72.70	68.83

Table 11: Waiting time and subnet utilization versus number of sites.

$class_{io}$–$prob$	ρ_d/ρ_c	$\overline{W}_{LOCAL}$	$\Delta \overline{W}_{X,LOCAL}/\overline{W}_{LOCAL}(\%)$		F_{LOCAL}	$\Delta F_{X,LOCAL}/F_{LOCAL}(\%)$	
			BNQ	LERT		BNQ	LERT
0.3	0.70	33.01	33.90	37.55	-0.377	76.66	73.74
0.4	0.81	28.63	39.78	42.71	-0.228	100.00	78.51
0.5	0.95	22.71	38.53	43.54	-0.042	-42.85	88.10
0.6	1.16	19.17	38.54	43.32	0.047	-76.60	-57.45
0.7	1.49	16.28	38.08	42.05	0.153	37.91	38.56
0.8	2.08	15.17	39.64	42.98	0.224	40.18	42.86

Table 12: $\overline{W}$ and F versus $class_{io}$–$prob$.

phenomenon, the subnet utilization at each *num_sites* setting is included in the table. This illustrates an important design consideration for distributed database systems — from the viewpoint of dynamic query allocation, there is an optimal value for the number of copies of data items. This value is in the range of 6-8 copies for the parameter settings used in this study.

To investigate the effect of dynamic allocation algorithm on the fairness of the system, a final experiment was performed in which the parameter $class_{io}$–$prob$ was varied from 0.3 to 0.8 (which makes the system without dynamic allocation favor one or the other of the query classes). The results are given in Table 12. Along with a significant waiting time improvement, an improvement in 'fairness' (measured in terms of $\Delta F_{X,LOCAL}/F_{LOCAL}$) is also evident. It is interesting that, no matter which class of query the system favors in the local case, dynamic query allocation tends to decrease the difference in normalized waiting times between the two classes.

6. CONCLUSIONS

6.1. Our Results

In this paper we have studied the problem of allocating queries to sites in a distributed database system. The particular environment that was studied was a fully-replicated distributed database system running two classes of queries. We have shown that significant decreases in the average waiting time for queries can be obtained when queries are allocated to sites based on load information. Another important result of this study is that the use of information about query resource demands leads to significantly better performance than a simple 'balance the number of queries' approach. This was shown in two ways, first through a simple analytical study, and then through a simulation study of several heuristic algorithms for assigning queries to sites. In addition, it was found that the overall fairness of the system can be improved as a result of dynamic query allocation.

6.2. Future Work

This study represents a first step towards incorporating load balancing techniques into a distributed database system. Our plans for the near future lie in two directions. First, we intend to investigate the possibility of moving partially executed queries from site to site at certain critical times, which will require determining when a query can be economically moved (probably between its primitive relational operations) and identifying the information that has to be transferred along with a query. This will lead to query allocation algorithms of a more dynamic nature. Second, we intend to address the general problem of dynamically allocating subqueries of distributed queries to sites in an environment with only partially replicated data. Our eventual goal is to integrate these ideas into an actual distributed query processing algorithm in order to demonstrate their feasibility.

We have seen that the benefits of dynamic task allocation are greater when the resource demands of tasks are considered. Database systems are not the only application in which I/O as well as CPU useage is important, nor is knowledge of the relative CPU and I/O needs of tasks restricted to the database environment. Many standard system utilities, such as compilers, text formatters, and file systems, have resource demands which could be characterized either statically or dynamically. This information could be used by a distributed operating system to make more informed load balancing decisions, as in our environment. The work reported in this paper is one stage of an ongoing study of how dynamic allocation strategies and task information can be used to improve the performance of distributed systems in general.

REFERENCES

[Aper83] Apers, P., Hevner, A., and Yao, S., "Optimization Algorithms for Distributed Queries", *IEEE Transactions on Software Engineering* 9(1), January 1983.

[Alon84] Alonso, R., *Query Optimization in Distributed Database Management Systems Through Load Balancing*, colloquium presented at the University of Wisconsin, Madison, April 1984.

[Bern81] Bernstein, P., et al, "Query Processing in a System for Distributed Databases", *ACM Transactions on Database Systems* 6(4), December 1981.

[Brya81] Bryant, R., and Finkel, R., "A Stable Distributed Scheduling Algorithm", *Proceedings of the Second International Conference on Distributed Computing Systems*, April 1981.

[DBE82] Special Issue on Query Optimization, *Database Engineering* 5(3), September, 1982.

[Epst78] Epstein, R., Stonebraker, M., and Wong, E., "Distributed Query Processing in a Relational Database System", *Proceedings of the ACM-SIGMOD Conference on Management of Data*, June 1978.

[Hevn79] Hevner, A., and Yao, S., "Query Processing in Distributed Database Systems", *IEEE Transactions on Software Engineering* 5(3), May 1979.

[Klei75] Kleinrock L., *Queuing Systems Volume II: Computer Applications*, Wiley, New York, 1976.

[Livn82] Livny M., and Melman M., "Load Balancing in Homogeneous Broadcast Distributed Systems", *Proceedings of the ACM Computer Network Performance Symposium*, April 1982.

[Livn83] Livny, M., *The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems*, Ph.D. Thesis, Weizmann Institute of Science, August 1983.

[McCo81] McCord, R., "Sizing and Data Distribution for a Distributed Database Machine", *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, May 1981.

[Melm84] Melman, M., and Livny, M., "The DISS Methodology of Distributed System Simulation", *Simulation*, April 1984.

[Ni81] Ni, L., and Abani, K., "Nonpreemptive Load Balancing in a Class of Local Area Networks", *Proceedings of the 1981 Computer Networking Symposium*, December 1981.

[Ni82] Ni, L., "A Distributed Load Balancing Algorithm for Point-to-Point Local Computer Networks", *Proceedings of COMPCON '82*, Fall 1982.

[Powe83] Powell, M., and Miller, B., "Process Migration in DEMOS/MP", *Proceedings of the 9th ACM Symposium on Operating Systems Principles, October 1983.*

[Reis78] Reiser M., and Lavenberg S.S., "Mean Value Analysis of Closed Multichain Queuing Networks", *JACM* 27(2), April 1980.

[Roth77] Rothnie, J., and Goodman, N., "A Survey of Research and Development in Distributed Database Systems", *Proceedings of the 3rd International Conference on Very Large Databases*, October 1977.

[Sacc82] Sacco, G., and Yao, S., "Query Optimization in Distributed Database Systems", in *Advances in Computer Systems*, Volume 21, Academic Press, New York, 1982.

[Seli79] Selinger, P., et al, "Access Path Selection in a Relational Database Management System", *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, June 1979.

[Seli80] Selinger, P., and Adiba, M., "Access Path Selection in Distributed Database Management Systems", *Proceedings of the First International Conference Conference on Distributed Data Bases*, Aberdeen, 1980.

[Yu83] Yu, C., and Chang, C., "On the Design of a Query Processing Strategy in a Distributed Database Environment", *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, May 1983.