LOAD BALANCING IN A LOCALLY DISTRIBUTED DATABASE SYSTEM

by

Michael J. Carey and Hongjun Lu

Computer Sciences Technical Report #625

December 1985

Load Balancing in a Locally Distributed Database System

Michael J. Carey
Hongjun Lu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

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

Load Balancing in a Locally Distributed Database System

Michael J. Carey
Hongjun Lu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

Most previous work on query optimization in distributed database systems has focused on finding optimal or near-optimal processing plans based solely on static system characteristics, and few researchers have addressed the problem of copy selection when data is replicated. This paper describes a new approach to query processing for locally distributed database systems. Our approach uses load information to select the processing site(s) for a query, dynamically choosing from among those sites that have copies of relations referenced by the query. Query compilation is used to produce a statically-optimized *logical plan* for the query, and then a dynamic optimization phase converts this logical plan into an executable *physical plan* at run-time. This paper motivates the separation of static and dynamic optimization, presents algorithms for the various phases of the optimization process, and describes a simulation study that was undertaken to investigate the performance of this approach. Our simulation results indicate that load-balanced query processing can provide improvements in both query response times and overall system throughput as compared to schemes where execution sites are either statically or randomly selected.

1. INTRODUCTION

1.1. Distributed Query Processing

Research in the area of distributed databases has led to the development of a number of algorithms for processing relational queries in distributed database systems. A recent survey paper by Yu and Chang [Yu84] reviews many of the distributed query processing algorithms that have been developed, and another recent survey article by Jarke and Koch [Jark84] reviews much of the research in query processing for centralized database systems (which is applicable for processing the local subqueries of distributed queries) and discusses some of the major issues involved in distributed query processing. One of the main distinguishing characteristics of distributed query processing algorithms is the cost models that they employ — the majority of algorithms proposed in the literature (including [Wong77, Hevn79, Chiu80, Bern81, Kamb82, Aper83, Yu83]) focus on minimizing communications cost-based metrics for processing a query, with only a few algorithms (such as [Epst78, Seli80]) considering both the local processing costs (e.g., I/O and CPU costs) and communications cost. The algorithms that concentrate on minimizing communications cost are usually intended for use in long-haul networks (as

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

opposed to local area networks), reducing the query optimization problem to that of finding optimal or near-optimal sequences of semijoins for processing various classes of queries.

One problem that has received relatively little attention is the problem of copy selection when data is replicated. Data replication is frequently cited as a way to improve the reliability and availability of distributed databases, but it complicates query optimization in that the data referenced by a query may be available at several candidate sites. A number of algorithms have simply ignored the problem, assuming either that data is not replicated or else that one copy of each item needed by the query, called a *materialization* of the database, is somehow selected prior to query optimization [Epst78, Hevn79, Bern81, Aper83]. Several query processing algorithms deal with replicated relations by considering all possible copies and choosing the copy of each relation that leads to the cheapest plan based on a combination of local processing and communications costs [Seli80, Liu82] (assuming that a single best copy of each relation indeed exists). Finally, an algorithm proposed by Yu and Chang uses a set of heuristics to choose a set of processing sites in the presence of replication (because the copy selection problem is NP-hard) [Yu83]. If communications cost is the primary concern, their algorithm chooses the minimum set of sites containing all of the relations referenced by the query; if local processing costs are significant, they suggest instead selecting the maximum set of sites to obtain parallelism. Virtually all distributed query processing strategies have been based entirely on static characteristics of distributed database systems (such as the topology and speed of the communications network or the physical organization of the data) as opposed to the dynamic characteristics of the system (such as the current workloads at each of the sites). Despite the amount of attention that has been focused on dynamic load balancing in distributed operating systems [Brya81, Livn82, Ni82, Eage85], little in the way of related work has been done in the distributed database area.

1.2. The Basis of Our Work

A locally distributed database system is one in which sites are connected via a low-latency, high-bandwith communications medium such as a token ring network or an Ethernet. This type of system has been proposed as being a good candidate architecture for a backend database machine, having the potential to provide high availability, good performance, and graceful expansion [McCo81, Ston85]. We have recently been investigating problems related to query processing in such a locally distributed database system.

In a recent paper [Lu85a], we reported on an experiment where we implemented and measured the performance of eight different distributed join algorithms in a testbed system based on the University of Wisconsin's Crystal multicomputer [DeWi84]. The algorithms that we investigated are eight algorithms resulting from selecting two options in each of three dimensions — primitive operator (join versus semijoin), processing paradigm (sequential versus pipelined execution), and local join algorithm (sort-merge versus nested loops with an index on the inner relation's join column). Briefly, the differences between these options for computing the equijoin $R_a[A=B]R_b$ using R_a as the outer relation are

as follows: Using the join as a primitive, tuples from R_a are sent to R_b's site and a traditional join algorithm is used to join tuples from the two relations there. In the semijoin case, the join column values $R_a.A$ are sent to R_b's site, where matching tuples are selected from R_b and returned to R_a's site where the join is completed. With sequential processing, entire temporary relations are shipped and stored, with the sites working serially. For example, R_a can be shipped and stored at R_b's site, after which a purely local join can be performed there. With pipelined processing, however, the join is computed using tuples as they arrive over the network, making it possible to avoid storing and re-scanning intermediate results and also providing some degree of parallelism. (One note here: When pipelined processing is used with semijoin-based methods, duplicate $R_a.A$ values are not eliminated before being sent, so a pipelined semijoin is basically the "fetch inner tuples as needed" method of System R* [Seli80].) Finally, the local join method options should be pretty much self-explanatory. Further details on these algorithms may be found in [Lu85a].

A number of experiments were performed for two-way join queries with varied source relation sizes, join selectivities, and join column value distributions. The results showed that, in a local network, communications cost is not the dominant cost factor. Shipping an entire relation from one site to another was found to be a reasonable way to process a distributed join, as long as the outer (smaller) relation was shipped to the inner site, and pipelining was used (to avoid storage and retrieval of a temporary relation). Pipelined processing outperformed sequential processing in nearly every test, except when there was a high join column duplication factor, in which case sequential semijoins outperformed pipelined semijoins due to the cost of re-accessing inner relation tuples for each duplicate $R_a.A$ value. The sort-merge methods performed best for joining two large relations, and the indexed nested loops methods performed best in all of the other cases examined. Finally, it was found that the combination of the join method and the join site is important, as they must be chosen to ensure that the outer relation is the smaller relation (the same as for centralized joins). The results were driven by local processing cost considerations in all cases [Lu85a].

In another recent paper [Care85], we studied the problem of selecting an execution site for a query in a distributed database system with fully-replicated data. We studied several dynamic query allocation algorithms, including *BNQ* ("Balance the Number of Queries"), where a newly arrived query is routed to the site with the fewest queries, *BNQRD* ("Balance the Number of Queries by Resource Demands"), where a newly arrived query is classified as being either CPU bound or I/O bound (based on information provided by the query optimizer) and is routed to the site with the fewest queries of the same class, and *LERT*, another scheme along the lines of *BNQRD*. This study showed several things. First, it was shown that, even if all sites have the same statistical workloads and processing capacities, load balancing can lead to significant reductions in the average time that queries spend waiting for CPU and I/O resources. Second, it was found that using information about queries, such as estimates of their CPU and I/O demands, improves the performance benefits provided by load balancing (i.e., that the *BNQRD* algorithm outperformed the simpler and more typical *BNQ* approach). This is of interest in database

systems because a query optimizer can provide such information.

Based on the results of our previous work, this paper proposes a new approach to query processing in a locally distributed database system with partially replicated data. The proposed approach uses load information to select the processing site(s) for a query, dynamically choosing from among those sites that have copies of relations referenced by the query. Query compilation is used to produce a statically-optimized *logical plan* consisting of a pipelined sequence of subqueries. A dynamic allocation phase then converts this logical plan into an executable *physical plan* at runtime. Section 2 motivates the separation of static and dynamic optimization and presents algorithms for the various phases of the optimization process. Section 3 describes a simulation study that was undertaken to compare the performance of this approach with that of schemes for allocating subqueries to execution sites either statically or randomly. Finally, Section 4 presents the main conclusions of this work.

2. LOAD-BALANCED QUERY PROCESSING

2.1. Some Possible Approaches

There are several ways that load balancing could be integrated with the query optimization and execution mechanisms of a distributed database system. One approach would be to integrate a dynamic query allocation algorithm like the BNQRD algorithm described in the previous section with an interpretive distributed query processing algorithm [Wong77, Epst78, Epst80], letting the query processing algorithm select the next subquery to execute, then using BNQRD to select from among the candidate sites for executing the subquery. The problem with this approach is that it would preclude query compilation, leading to an unacceptable runtime overhead for query optimization. A second possible approach would be to compile 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 [Alon84]. This approach is attractive in terms of runtime overhead, but it complicates query optimization — both the storage sites of relations and the various possible system load conditions would have to be considered during query compilation, and the problem of identifying load conditions of interest may be difficult. In addition, the number of alternative plans that can be generated will be limited by storage space and by the complexity of the query (since multi-relation queries may lead to a large number of alternative plans). The approach that we advocate for integrating load balancing and query processing is quite different than these two possibilities.

2.2. The LBQP Approach

Our approach involves doing static planning at compile time followed by dynamic allocation at runtime. Figure 1 outlines this distributed query processing scheme, which we call the LBQP (Load-Balanced Query Processing) algorithm for distributed query processing.

As shown in Figure 1, the LBQP algorithm consists of three phases. The first phase of LBQP is the *static planning phase*, which is the compilation phase of the algorithm. The input to this phase is a

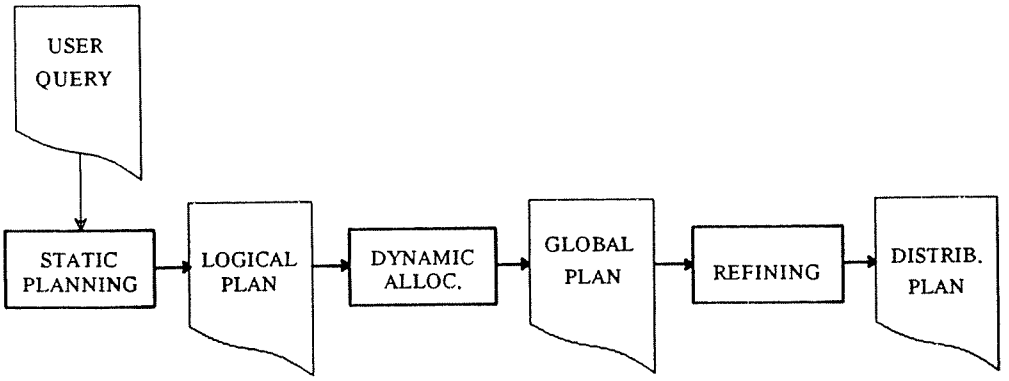

Figure 1: Algorithm LBQP — Static Planning + Dynamic Allocation.

relational query, and its output is a *logical processing plan*, a sequence of relational operations on logical relations (relations whose physical locations have not been bound yet). The local processing costs for each relation and the names of the sites where each relation is stored are attached to the plan for later use. The second phase of LBQP is the *dynamic allocation phase*, which is invoked at runtime. The input to this phase is the logical plan, and its job is to select a physical copy (i.e., site) for each relation in the plan, producing a *global processing plan* as its output. The final phase of LBQP is the *refining phase*, which chooses between join-based and semijoin-based pipelined distributed join methods for each join in the global plan that involves relations at two different sites. An advantage of the LBQP approach is that it simplifies query optimization, as static planning can be performed without considering load information (making a number of existing optimization algorithms reasonable candidates for the static planning phase). This also allows most of the query optimization work to be done once, during the static planning phase, with only the dynamic allocation and refinement algorithms being invoked each time the query is to be executed. We now consider each phase of the LBQP algorithm in turn.

2.2.1. Static Planning

As described above, the job of the static planning phase of the LBQP algorithm is to generate a statically optimized processing plan for a query. Since this phase is independent of the dynamic allocation phase, it would be possible to use most any existing distributed query processing algorithm in this phase, producing a logical plan where the set of candidate processing sites for each relation are those sites for which the plan has the cheapest (static) overall cost. However, based on our previous performance results for join processing in locally distributed database systems [Lu85a], which we summarized in the preceding section, the static planning phase can be further simplified. The LBQP scheme makes the following set of simplifications:

(1) *For two-way joins, only a set of local join methods plus their pipelined distributed counterparts are considered for processing join queries.* This simplification is justified directly by the results reported

in [Lu85a], where we found that pipelined distributed join methods performed better than sequential methods over a wide range of distributed join queries.

(2) *For n-way joins, only linear sequences of two-way joins are considered during static planning.* This heuristic is used in System R [Seli79] and System R* [Seli80], and limits the optimizer's search place considerably with the risk of missing the optimal order in only a few cases [Lohm85]. The resulting plans are easily executed in a pipelined fashion, with intermediate relations only being stored before sort operations (if any).

(3) *Local join methods (e.g., nested loops or sort-merge) are selected without considering where relations are stored.*[1] For local joins, this is obviously reasonable. For distributed joins, it is still reasonable, as the local join method does not affect the amount of data transferred during the join.

(4) *Inner and outer relations are chosen without considering where relations are stored.* Our experimental results indicate that the choice of inner and outer relations for locally distributed joins should be the same as for local joins [Lu85a]. Pipelined join-based methods and their semijoin-based counterparts were found to perform similarly, the difference being that the result relation ends up at the inner site with join-based methods and at the outer site using semijoin-based methods; thus, the site where the join's result will end up can be controlled by choosing between joins and semijoins (rather than switching to a sub-optimal choice for the inner and outer relations).

Given these simplifications, the static planning phase can optimize a query as though it were going to be executed in a centralized database system containing the logical relations (and their access paths) referenced by the query. In LBQP, the objective of the static planning phase is to minimize the total local processing cost for the query, including both the I/O and CPU costs. Thus, an optimizer like the access path selector of System R [Seli79, Lohm85] can be used to generate the logical plan. Since the design of such optimizers is well-understood, we do not discuss the design of the static planning phase further. The output of this phase is a plan that specifies the join order for the query, the local join method and the inner and outer relations for each join, the access paths to be used when accessing the relations, and the information required by the dynamic allocation phase (the set of storage sites for each relation, the estimated CPU and I/O costs for accessing each relation, and the estimated result sizes).

2.2.2. Dynamic Query Unit Allocation

This phase of the LBQP algorithm is responsible for selecting copies of each of the relations referenced in a query plan. This phase views a logical query plan as a sequence of (pipelined) processes, each of which requires a certain amount of CPU and I/O service and communicates with its neighboring processes. Each process in the plan is actually a *query unit*, the maximum processing unit that accesses a single relation, and the plan is treated as an ordered list of query units. (Note that each half of a join

[1] We assume here that all copies of a relation have the same set of access paths available, although this assumption can be relaxed [Lu85b].

is thus a query unit.) The job of dynamic allocation is thus to select execution sites for each of the query units in the list — we refer to this as the *query unit allocation problem*. This problem can be stated more formally as follows.

In the BNQRD algorithm [Care85], queries (query units in this case) are classified as being either I/O bound or CPU bound. The load of a site s_j is expressed as the number of I/O bound and CPU bound query units at the site. That is, $LD_{BNQRD}(s_j) = \{ NQ_{IO}(s_j), NQ_{CPU}(s_j) \}$. The objective is to balance the CPU and I/O loads of the sites in the system, so the balance of the system for a given allocation of query units to sites can be measured as the variance of the I/O and CPU loads of the sites using the following *BNQRD unbalance factor*:

$$UBF_{BNQRD} = \frac{\sum_{j=1}^{n} ((NQ_{IO}(s_j) - \overline{NQ}_{IO})^2 + (NQ_{CPU}(s_j) - \overline{NQ}_{CPU})^2)}{n}$$

$\overline{NQ}_{IO}$ and $\overline{NQ}_{CPU}$ are the average numbers of I/O and CPU bound queries for the sites and n is the number of sites in the system.

For a locally distributed relational database with n sites, $\{s_1, \ldots, s_n\}$, the query unit allocation problem can then be expressed as follows:

Given:

(1) A user query Q expressed as a sequence of query units, $Q = \{q_1, \ldots, q_m\}$, each known to be either I/O bound or CPU bound (from their estimated processing costs).

(2) A feasible assignment set $S_i = \{s_{i_1}, \ldots, s_{i_k}\}$ for each unit q_i, $0 \leq i \leq m+1$, containing the sites with copies of the relation that q_i references. (Query units q_0 and q_{m+1} are dummy query units representing the sites where the query is initiated and where its results are to be sent; they have no processing costs, and just one site in each of their feasible assignment sets.)

(3) An *initial load vector* specifying the initial load at each site s_j, $1 \leq j \leq n$, as given by $LD_{BNQRD}(s_j)$.

Find:

An optimal allocation of query units to processing sites such that:

(1) The unbalance factor under this allocation is minimized.

(2) The total communications cost, measured as the number of nonlocal query unit pairs in the sequence (including the two dummy query units), is also minimized to the extent possible without increasing the unbalance factor of the allocation. (A nonlocal pair is a pair of adjacent query units q_i and q_{i+1} that are allocated to different sites.)

This allocation problem is fairly unique (as compared with other task allocation work) in that it uses a quantitive measure of the degree of balance of the system's load, it takes the initial system load into account, it attempts to balance both the CPU and I/O loads, and it also attempts to minimize the communications cost by choosing the balanced allocation with the smallest number of nonlocal pairs. As a result, typical solution methods such as integer programming are inappropriate for solving this problem [Lu85b], and an exhaustive solution would be too slow (running in exponential time). Instead,

we have developed an algorithm that finds near-optimal solutions in a reasonable amount of time.

Our dynamic query unit allocation algorithm employs several heuristics to aid it in making good allocation decisions. One heuristic controls the order in which query units are considered for allocation to sites — query units with less flexibility about where they can run are allocated earlier than query units with greater flexibility. We define the *static freedom* of a query unit to be the number of sites in its feasible assignment set, and we define its *dynamic freedom measure* to be the sum of the loads of these sites. (The load of a site with respect to query unit q_i under BNQRD is the number of query units there of the same class as q_i.) The algorithm considers queries in increasing order of static freedom, allocating query units with only a few candidate sites earlier than query units with many. In the event of a tie, the query unit with the largest dynamic freedom measure is chosen because its candidate sites are more heavily loaded (giving it fewer desirable site choices).

The other two heuristics control the sites to which query units are allocated. The first heuristic eliminates heavily loaded sites from consideration — any site with a class load that exceeds the average class load expected after allocation is considered to be full for that class, and is thus deleted from the feasible assignment set of all query units of that class (as long as they have at least one other site in their feasible assignment set). The other heuristic is a bit more complex — to select a site for a query unit q_i, the *current load* of each site is considered first, the *benefit* is considered next (in the event of a tie), the *potential load* is considered after the benefit (in case of another tie), and the *potential benefit* is considered last. The current load of a site is the number of query units of q_i's class there, as usual for BNQRD. The benefit of a site is the number of adjacent query units (q_{i-1} or q_{i+1}) that have already been allocated to the site (either 0, 1, or 2), serving as a measure of the communications cost savings associated with allocating q_i to the site. The potential load of a site is the number of unassigned query units of q_i's class that have the site in their feasible assignment sets, which is the additional load that could be introduced by the allocation process. Finally, the potential benefit of a site is the number of unassigned query units whose feasible assignment sets include the site and that would form a consecutive local run of query units (including q_i) if they were assigned to the site; this is a measure of the possible communications cost savings due to allocating q_i to the site.

With these heuristics, the basic algorithm for dynamic query unit allocation can now be presented. The complexity of the algorithm turns out to be $O(\max(mn, m^2 \log_2 m)$ because it considers each query unit in turn, potentially considering every site for the current query and also re-sorting the list of remaining queries once the current query has been allocated to a site. (Remember that m is the number of query units, so it will be fairly small — for a 3-way join query, for example, m will be 5 with the two dummy query units included.)

Load-Balanced Dynamic Query Unit Allocation:

(1) Allocate q_0 to the query's site of origin, and allocate q_{m+1} to the result site.

(2) Compute the static and dynamic freedom metrics for each query unit q_i, $1 \leq i \leq m$.

(3) Select the next query unit to be allocated as the one with the least assignment flexibility using the freedom metrics.

(4) Select an allocation site s_j for this query unit by choosing the site with the least (BNQRD) load, considering the benefit, potential load, and potential benefit metrics in turn as necessary.

(5) Increment the load information for class q_i queries for site s_j, and recompute the freedom metrics of any unallocated query units of q_j's class that have s_j in their feasible assignment set. (If s_j's load for the class has exceeded the expected average, simply delete s_j from their feasible assignment sets if possible.)

(6) If any unassigned query units remain, go to step 3 and repeat the process.

In order to investigate the reasonableness of this heuristic algorithm, we implemented it and compared it to an exhaustive search for a large number of randomly generated queries [Lu85b]. In these tests, we varied the number of real (as opposed to dummy) query units in the query from 3 to 6, the number of sites in the system from 4 to 12, the average number of copies of relations referenced by the query units from 2 up to the total number of sites, and we also varied the initial unbalance of the system. It was found that the algorithm finds allocation plans with the optimal unbalance factor about 95% of the time, but that its frequency of producing optimal allocations with respect to communications cost is somewhat worse (often on the order of 80-90%, and only about 25% in the case of fully-replicated data). The running time for the algorithm (unoptimized on a VAX 11/780 under Unix[2]) was found to be in the range of 17-40 milliseconds when 8 sites contain data relevant to the query; the times for 3 query units were in the 17-24 millisecond range, and for 4 query units the range was 24-31 milliseconds. This is fast enough so that dynamic query unit allocation will not add an undue amount of runtime overhead.

As a result of these tests, two enhancements were added to the basic algorithm in the form of post-processing routines which try to lower the communications cost for the allocation plan without altering its unbalance factor. The first enhancement considers each query unit q_i, checking each site in its feasible assignment set to see if the query unit could instead be placed there without changing the unbalance factor but yielding a reduction in communications cost. The second enhancement looks at each pair of adjacent query units (q_i, q_{i+1}) and tries to find a third query unit q_j that can be exchanged (site-wise) with q_{i+1} so as to cut down the communications cost (again without affecting the unbalance factor in the process). The complexities of these enhancements are $O(nm)$ and $O(m^2)$, respectively, so they do not increase the overall complexity of the basic algorithm. They were found to significantly improve the performance of the algorithm in terms of communications cost optimality, actually allowing it to always find the optimal plan in the fully-replicated case, while adding only a few milliseconds to the dynamic allocation process.

---

[2] Unix is a trademark of AT&T Bell Laboratories.

2.2.3. Refining the Plan

The third and final phase of the LBQP algorithm is the refining phase. As described at the beginning of this section, LBQP considers only pipelined distributed versions of local join methods for computing distributed joins. The input to the refining phase is the processing plan produced by the static planning phase augmented with the site information produced by the dynamic allocation phase (i.e., for each relation, the name of the site where the query unit that accesses the relation will execute). For each distributed join in the plan, the refining phase decides whether the query units involved in the join (q_i and q_{i+1}) will use a pipelined join-based algorithm or a pipelined semijoin-based algorithm. In the former case, the result will end up at q_{i+1}'s site, and in the latter case it will end up at q_i's site. The decision is made based on the communications cost of each alternative — these costs are computed from the size estimates attached to the processing graph. (Note that this is the only phase that considers the communications cost at this level of detail, as the previous phase simply viewed it as being present or not.)

The communications cost for a pipelined join is the cost of sending all of relation R_{outer} to the site of R_{inner} for processing, plus the cost of sending the tuples produced by the join to the site of the next query unit (if it is not at R_{inner}'s site). For a pipelined semijoin, the cost consists of sending the join column value of each R_{outer} tuple to the inner site, plus sending back the matching R_{inner} tuples, plus sending the result tuples to the next processing site (if necessary). The refining phase simply considers each nonlocal join in the processing graph, one at a time, computes these two alternative costs, and chooses the cheaper of the two distributed join execution paradigms for each one. It attaches the result of its decision to the processing plan, which can then be translated into executable form and distributed to the sites for execution.

While queries could be processed completely using only pipelined join-based methods, the use of semijoin-based methods provides an opportunity to reduce the overall communications cost without changing the balance of the system.[3] An example of the type of communications cost savings possible in the refining phase is as follows: Consider a join $R_a[A=B]R_b$ in a system with three sites, S_1, S_2, and S_3. Suppose that the query originates at site S_1 and that the results are to be returned to the user at this site. Also, assume that the allocation phase decides to use the copies of R_a at S_1 and R_b at S_2 for load balancing reasons. If a join-based algorithm is used, the result will be produced at S_2, and the resulting tuples will have to be piped back to S_1. If a semijoin-based method is used instead, the result can be accumulated directly at S_1, avoiding this final communications cost.

---

[3] We assume that the CPU cost of merging tuples from the relations being joined is small enough compared to the cost of accessing the tuples that moving the merging responsibility from one query unit to the other will not change their classes (e.g., I/O bound or CPU bound).

3. A PERFORMANCE STUDY

The best way to evaluate the performance gains available using the LBQP scheme would be to test the ideas out in a real distributed database system. This would require the use of a distributed multi-user DBMS, however, and our own distributed join testbed [Lu85a] is only a single-user testbed. Instead, we conducted a simulation study to investigate the effectiveness of our dynamic query unit allocation algorithm. In this study, we compared the LBQP approach to several previously proposed allocation schemes under a number of different assumptions about the workloads at the sites and the level of data replication in the system. This section describes our simulation model, the other allocation algorithms studied, the metrics used to evaluate the performance of the algorithms, and the results of several experiments.

3.1. A Simulation Model

3.1.1. Structure of the Model

Our model of a distributed database system consists of a set of database sites (DB sites) plus a local area communications network. Figure 2 depicts our DB site model. Solid lines indicate the flow of query units, and dotted lines illustrate the flow of intermediate result data. A user query is represented as a sequence of query units, and each query unit runs at one site, accesses one relation, and passes data on to the next query unit in the query. This model has been implemented as a simulation program using the Modula-2-based DENET simulation language [Livn85].

When a query is initiated by a terminal, the BNQRD-based dynamic query unit allocation algorithm is applied to decide the processing site for each of its query units. If a query unit is allocated to a site other than its originating site (called its *home site*), it is transferred to its processing site via the network. Since all query units in a query are processed in a pipelined fashion, only the first query unit in the pipeline begins execution right after the query has been allocated. The other query units will enter the *blocked queue* and will not begin executing until receiving the first data page from their predecessors in the pipeline. For each data page received, a query unit will cycle through the disk and CPU service centers several times. Upon finishing this process, the query unit then checks the data pool. If the next data page is already in the pool, the query unit gets the page and continues executing. Otherwise, the query unit will either block awaiting the arrival of the next input data page, or else it will terminate (if its predecessor has completed).

After each cycle through the disk and CPU servers, a query unit produces a certain amount of (intermediate) result data. When one page of result data has accumulated, an outgoing data page is formed. If the query unit that produced the data is the last one in the pipeline, the data page is directed back to the terminal at the query unit's home site; otherwise, the page is passed to the next query unit in the pipeline. Depending on the site of the next query unit, the data page will either be sent to the data pool at a remote site (via the network) or placed in the local data pool.

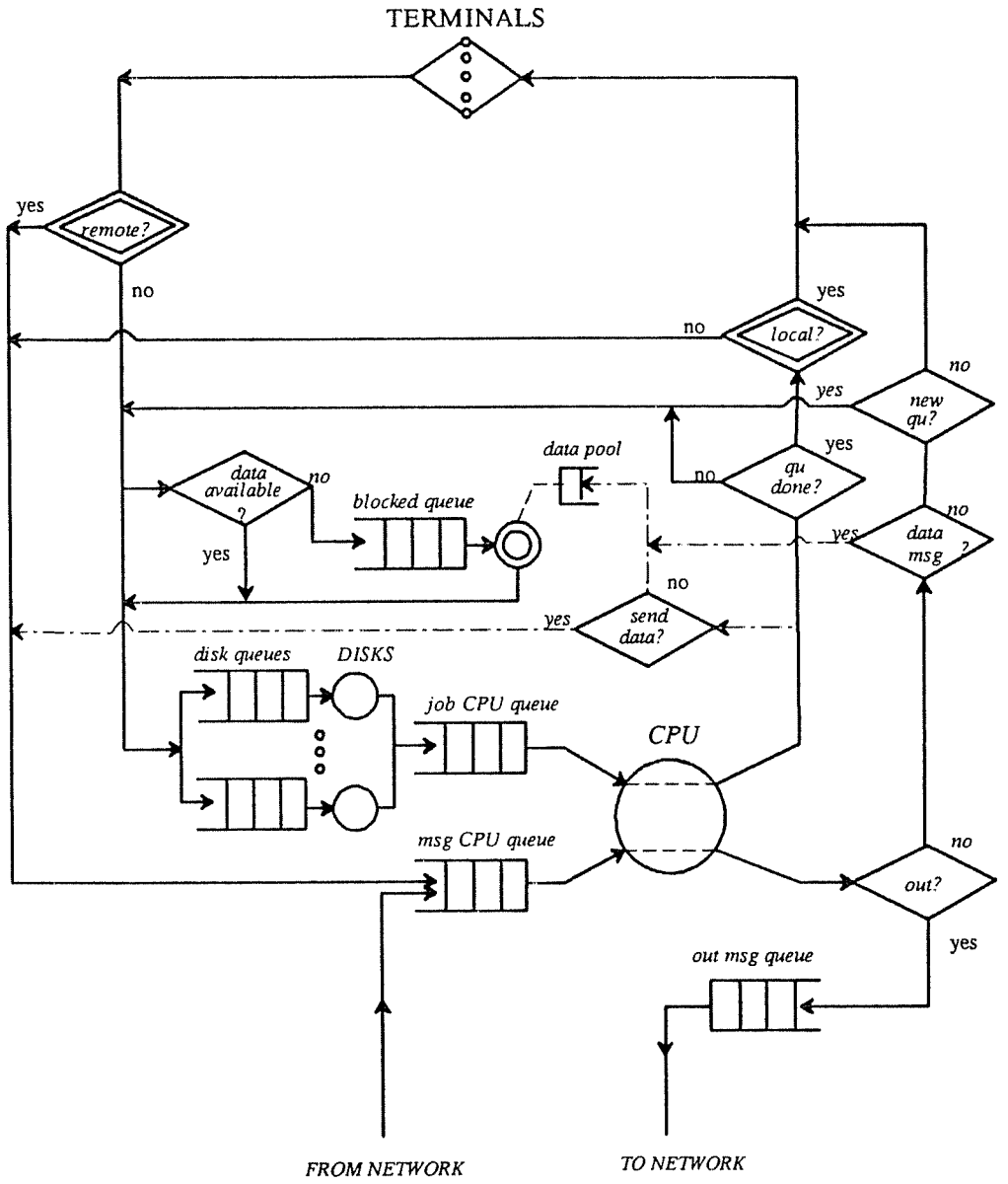

Figure 2: The Database Site Model.

The model includes two types of messages. The first is a *query message*, used to carry information about a query unit to a remote site for execution (or a completion signal to the home site when the last query unit finishes). The other type of message is a *data page message*. Both incoming and outgoing messages first enter the *message CPU queue* to receive CPU service for protocol-related processing; an outgoing message then enters the *out message queue* to be served by the communications subnet. If an incoming message is a data message, it is directed to the data pool; if it is a query message it is routed either to the CPU, to the disk, or else back to the appropriate terminal if it is a query completion message. The network model is the same as that of [Care85], a simple model of a token ring network that

considers requests from ready sites in a round-robin fashion and allows only one message at a time to actually be on the network. Lastly, we ignore the issue of how load information is exchanged among the sites (perhaps assuming that it is "piggybacked" on other messages), as this issue has been addressed by others [Brya81, Livn82, Ni82, Eage85].

3.1.2. Model Parameters

Tables 1 through 3 give the system parameters, the DB site parameters, and the communications-related parameters for the model. These parameters should be fairly self-explanatory, with the possible exception of the parameter $storage\_sites_k$ — this parameter set gives the storage sites for each relation in the database.

The system's workload is described by three groups of parameters, query parameters, query unit parameters, and class parameters, as shown in Table 4. Queries are classified into *query types* based on their number of query units, and the parameter *num_qtypes* is the number of different query types in the system. For each query type i, num_qus_i specifies the number of query units for this query type, and $qtype\_prob_i$ is the probability that a new query will be a type i query. For each query unit j of a type i query, the query unit parameters $num\_reads_{i,j}$, $res\_fraction_{i,j}$ and $class_{io}\_prob_{i,j}$ describe its service

System Parameters	
num_sites	number of DB sites in the system
num_rels	number of relations in the database
$storage\_sites_k$	sites with a copy of relation $R_k, 1 \leq k \leq num\_rels$

Table 1: System Parameters.

DB Site Parameters	
num_disks	number of disks per DB site
disk_time	mean disk access time per page
disk_time_dev	percent variation in disk time
mpl	number of terminals per DB site
think_time	mean terminal think time

Table 2: DB Site Parameters.

Communications-Related Parameters	
msg_cpu_rate	message CPU time per message byte
msg_setup	fixed network time for message initiation
trans_rate	network time per byte of data
query_descrip_size	size of a query message in bytes
data_page_size	size of a data page message in bytes

Table 3: Communications-Related parameters.

Query Parameters	
num_qtypes	number of query types
num_qus_i	number of query units for query type i
$qtype\_prob_i$	probability of a query being a type i query ($1 \leq i \leq num\_qtypes$)
Query Unit Parameters for Query Unit j	
$num\_reads_{i,j}$	mean number of reads for query unit j
$res\_fraction_{i,j}$	mean fractional result size for query unit j
$class_{io}\_prob_{i,j}$	probability that query unit j is I/O bound ($1 \leq i \leq num\_types$; $1 \leq j \leq num\_qus_i$)
Query Unit Class Parameters	
$page\_cpu\_time_{io}$	mean per-page CPU demand for I/O-bound query units
$page\_cpu\_time_{cpu}$	mean per-page CPU demand for CPU-bound query units

Table 4: Workload Parameters.

demands. The parameter $num\_reads_{i,j}$ defines the mean number of cycles through the CPU and I/O service centers for the query unit. For the first query unit ($j = 1$), this is the number of pages it reads from the disk. For the other query units ($j > 1$), since their execution is dependent on the intermediate result data from their predecessors, $num\_reads_{i,j}$ is the mean number of processing cycles (exponentially distributed) corresponding to the receipt of one data page from the preceding query unit. The parameter $res\_fraction_{i,j}$ specifies the fraction of a result data page generated from each page processed. Finally, query units are classified into two classes, I/O bound and CPU bound, according to their I/O and CPU service demands, as in [Care85]. The parameter $class_{io}\_prob_{i,j}$ is the probability that query unit j will be I/O bound. The $page\_cpu\_time_{io}$ parameter gives the mean CPU time (exponentially distributed) required to process a page of data for I/O bound queries, and $page\_cpu\_time_{cpu}$ gives the mean page CPU time for CPU bound queries.

3.2. Other Allocation Algorithms

The main purpose of this simulation study was to examine the performance of LBQP's dynamic query unit allocation algorithm. For comparison purposes, the following other query unit allocation algorithms were also studied:

STATIC. This algorithm processes a query unit at its predecessor's processing site if possible. If the referenced relation is unavailable at the local site, a statically predetermined copy of the relation (i.e., its primary copy site) is selected. This algorithm was motivated by existing distributed query processing algorithms which use a predetermined copy for query optimization [Bern81].

RANDOM$_F$. This algorithm randomly selects a processing site for an entire query when it is initiated. All of its query units will be processed at that site. (This is for fully replicated data only.)

RANDOM$_P$. Algorithm RANDOM$_P$ is a random site selection scheme for partially replicated data, selecting a processing site for a query unit as follows: A query unit is allocated to the site of its predecessor if possible (to minimize communications cost). If the referenced relation is unavailable at that site, a site is selected at random from among the sites with copies of the relation. This algorithm is similar in some respects to the allocation schemes of [Seli80, Yu83, Wang85] assuming that all copies of the relations have the same access paths.

3.3. Parameter Settings and Performance Metrics

The parameter settings used in the experiments reported here are given in Table 5. The system consists of 6 sites. There are 3 relations in the database, so the number of query units in a query varies from 1 to 3. The number of query types is also varied from 1 to 3. The number of terminals at each site, the *mpl*, is 5. Since the maximum number of query units for a query is 3, the number of actual processes running at a site may reach 15. The *num_reads* setting is 20 pages for the first query unit and 5 for the others. With a *res_fraction* of 0.2, the mean number of result pages for the first query unit is 4; therefore, the total number of reads for the second and third query units is also 20. (All of the query types have the same query unit characteristics here.) In this study, the mean value of *page_cpu_time* is assumed to be 4 times the *disk_time* for the CPU bound query units and 1/4 of the *disk_time* for I/O bound query units. The settings for the remaining parameters were chosen to model a locally distributed system similar to that of our join experiments [Lu85b]. In each experiment, the *think_time* parameter is varied to yield different system loads.

System Parameters		**Query Parameters**	
num_sites	6 sites	*num_qtypes*	1-3
num_rels	3 relations	*num_qus*	1-3
storage_sites	(varied in tests)	*qtype_prob*	(varied in tests)
DB Site Parameters		**Query Unit Parameters**	
num_disks	2 disks	*num_reads*	20, 5, 5
disk_time	20 msec	*res_fraction*	0.2, 0.2, 0.2
disk_time_dev	±20%	*class_prob*	0.5, 0.5, 0.5
mpl	5 terminals		
think_time	1.0 - 28.0 sec		
Communications Costs		**Query Unit Class Parameters**	
msg_setup	250 μsec	$page\_cpu\_time_{io}$	5 msec
msg_cpu_rate	2.0 μsec/byte	$page\_cpu\_time_{cpu}$	80 msec
trans_rate	2.0 μsec/byte		
query_descrip_size	2048 bytes		
data_msg_size	4096 bytes		

Table 5: Simulation Parameter Settings.

In order to evaluate and compare the performance of the various query unit allocation algorithms, several performance metrics will be used. Two of these are the average response time for queries, measured as the difference between their submission and completion times, and the throughput of the system in queries per second. In addition, load balancing is intended to reduce the amount of time that query units spend waiting for service in the CPU and disk queues. In order to measure the waiting time improvement provided by a query unit allocation algorithm in this environment, the best-case average response time of a query mix is obtained by simulation (by executing all query units of a query concurrently at different sites in "single-user mode"); this time is denoted by R_{su} (where "su" stands for "single user mode"). We then define the mean waiting time improvement factor of algorithm L_A over algorithm L_B [Care85] as follows here:

$$WIF(L_A, L_B) = \frac{(R_B - R_{su}) - (R_A - R_{su})}{R_B - R_{su}} = \frac{R_B - R_A}{R_B - R_{su}}$$

R_A and R_B are the mean response times when allocation algorithms L_A and L_B are used, respectively. In particular, to compare an algorithm L to the STATIC allocation algorithm, $WIF(L, STATIC)$ will be used.

3.4. Experiments and Results

Three experiments were performed to investigate the performance of the four query unit allocation algorithms LBQP, RANDOM$_F$, RANDOM$_P$, and STATIC. Experiment 1 examines the performance of the algorithms as the degree of replication of the relations was varied. Experiment 2 examines the benefits of load balancing in a situation where some sites have heavier workloads (rates of arriving queries) than other sites. Finally, Experiment 3 looks at how the query allocation algorithms perform under a workload consisting of a mix of queries with different numbers of query units.

Test Query for Experiment 1			
# of Query Units	3		
Query Unit	1	2	3
referenced relation	R_1	R_2	R_3
num_reads	20 pages	5 pages	5 pages
res_fraction	0.2	0.2	0.2
page_cpu_time	80 msec	5 msec	5 msec

Table 6: Test Query for Experiment 1.

Relations and Their Storage Sites				
Site	Case 1	Case 2	Case 3	Case 4
$S1$	$\{R1,R2,R3\}$	$\{R1^*,R3\}$	$\{R1^*\}$	$\{R1\}$
$S2$	$\{R1,R2,R3\}$	$\{R1,R3\}$	$\{R1\}$	$\{R1\}$
$S3$	$\{R1,R2,R3\}$	$\{R1,R2^*\}$	$\{R2^*\}$	$\{R1,R2^*\}$
$S4$	$\{R1,R2,R3\}$	$\{R1,R2\}$	$\{R2\}$	$\{R1,R2\}$
$S5$	$\{R1,R2,R3\}$	$\{R2,R3^*\}$	$\{R3^*\}$	$\{R1,R3^*\}$
$S6$	$\{R1,R2,R3\}$	$\{R2,R3\}$	$\{R3\}$	$\{R1,R3\}$
(*'s denote primary copies of partially replicated relations)				

Table 7: Relations' Storage Sites.

3.4.1. Experiment 1: Degree of Data Replication

Queries in this experiment have three query units, the first being a CPU bound query unit and the others being I/O bound. This corresponds roughly to a 3-way join where the first relation is scanned sequentially and the other relations are accessed through a nonclustered index. The service demands of the three query units are listed in Table 6, and the other parameters are those listed in Table 5. The relations in the database and their storage sites are given in Table 7 — this experiment consists of four tests, each with a different level of data replication. In Test 1, all three relations are replicated at every site (Case 1 in Table 7). In Test 2, there are four copies of each relation (Case 2), and there are two copies of each relation in Test 3 (Case 3). In Test 4, the most heavily used relation (R_1) is fully replicated, and the other two relations each have only two copies (Case 4).

Figure 3 presents the results of Test 1, where all three relations are fully replicated. The mean response time and the waiting time improvement factor *WIF(LBQP,STATIC)* are given as functions of think time. Since the data is fully replicated and the workloads are the same at each site, all sites have the same CPU and disk utilizations; the CPU utilization is given in Table 8 for reference. The measured lower bound for the mean query response time is also shown in Figure 3.

The first thing to note from Figure 3 is that the load-balanced query unit allocation algorithm (LBQP) leads to a significant decrease in waiting time with respect to the STATIC algorithm except when sites are heavily loaded, showing that it is indeed better to place the query units of a query at different sites for load balancing purposes than to always process them locally (which is usually assumed to be the right thing to do). When the *think_time* is 16 seconds or more, which corresponds to a CPU utilization of less than 0.5 at each site, the waiting time improvement factor is over 50%. As the system becomes more heavily loaded, the improvement becomes smaller. With 4 seconds of *think_time*, where the CPU utilization is over 0.9, LBQP actually performs slightly worse than STATIC. This is because the result fraction is fairly large (0.2), so a large intermediate result (about 4 pages per query unit) must be transferred when the query units are nonlocally allocated (which consumes CPU time for message processing). A second observation from this test is that using a random scheme for selecting processing

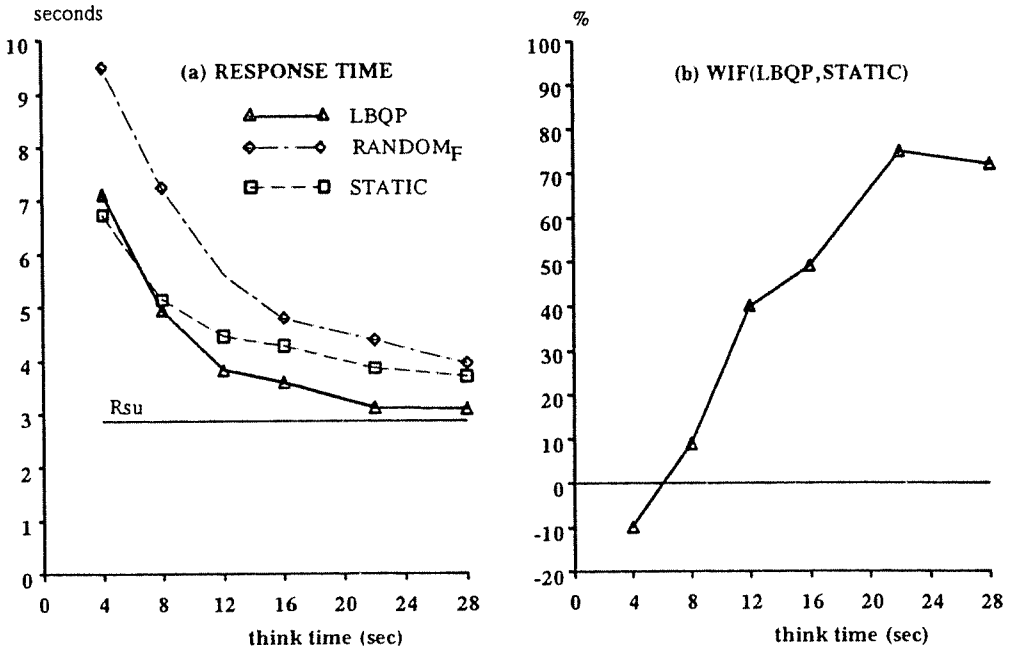

Figure 3: Results of Experiment 1, Test 1 (full replication).

CPU Utilization in Experiment 1, Test 1							
Think Time	4	6	8	12	16	22	28
CPU Utilization	0.92	0.82	0.73	0.59	0.50	0.38	0.31

Table 8: CPU Utilization in Experiment 1, Test 1.

sites is not advisable. The response time for RANDOM$_F$ is always larger than that of STATIC in Figure 3. (*WIF*(*RANDOM*$_F$,*STATIC*) is negative and is not shown in Figure 3.) The reason for this is that a randomly selected remote site is just as likely to be heavily loaded as the local site, so choosing a non-local site only increases the communications cost.

For the throughput (which is not shown), we found that RANDOM$_F$ was again worse than STATIC, but that LBQP only increased throughput over STATIC by a few percent in this case. The reason for this small gain is that if the system is lightly loaded, its throughput is limited by the query arrival rate, so LBQP's response time improvements in this region do not help much in terms of throughput. The throughput is determined by the waiting time of the queries only under heavy loads, and LBQP did not improve the mean waiting time under heavy loads in this experiment.

Tests 2 and 3 of Experiment 1 examined the performance of the query unit allocation algorithms with partially replicated data. There were four copies of each relation in Test 2, and in Test 3 each relation had two copies. (The distribution of copies is shown in Cases 2 and 3 of Table 7.) The results are given in Figures 4 and 5. Throughput results are included this time, with both throughput itself and the

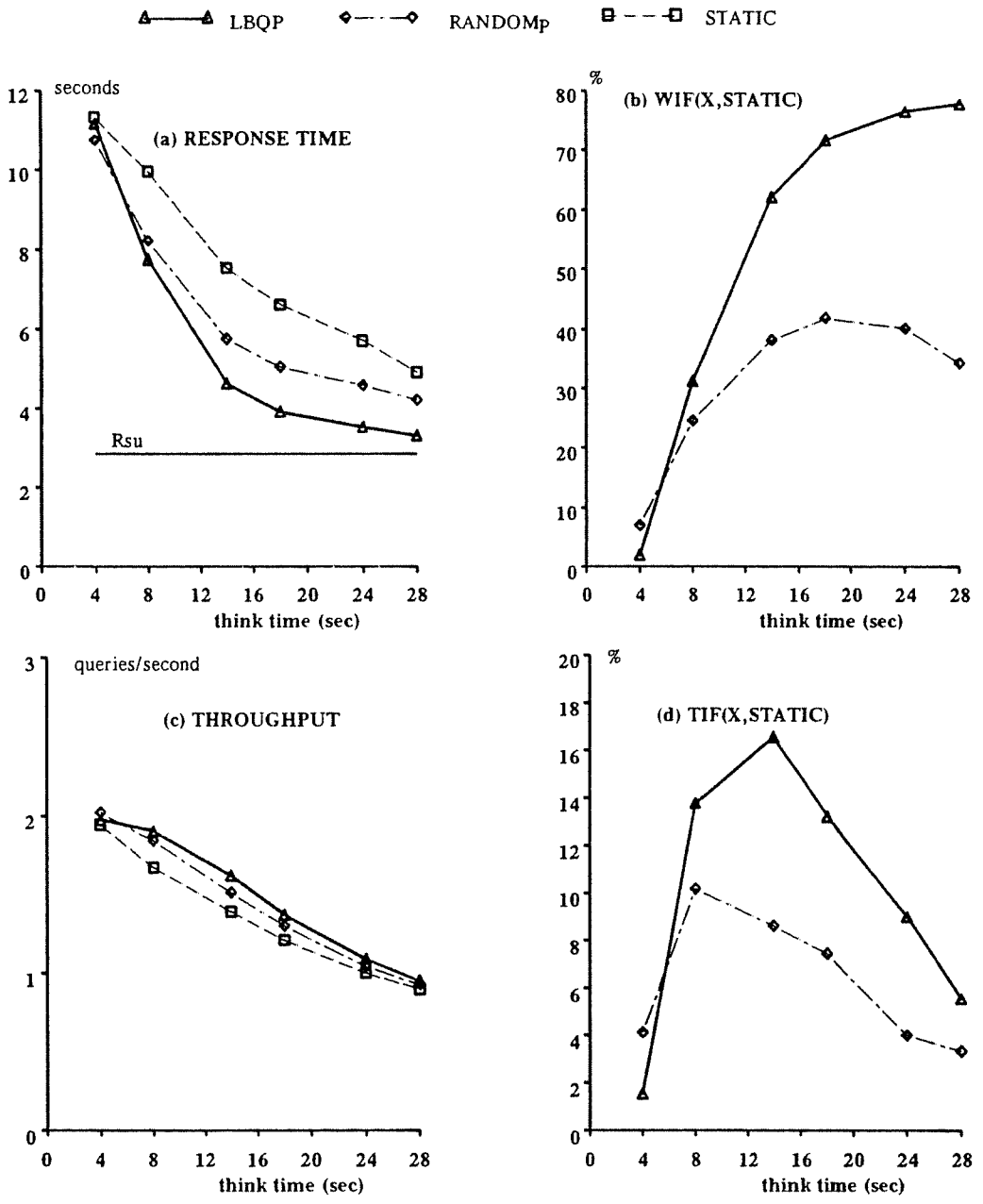

Figure 4: Results of Experiment 1, Test 2 (4 copies/relation).

throughput improvement as compared to STATIC allocation being given.

Several observations can be made from these results. First, the response time for STATIC allocation increases as the number of copies is decreased, especially when the *think_time* is short. This is caused by increased resource contention at the sites storing primary copies, which is exactly the problem that LBQP is intended to alleviate. It is evident that LBQP lowers the response time significantly in both tests, and also that the waiting time improvement factor increases with decreasing system load. The rate of this increase is a little higher in the case of four copies per relation, as the greater number of copies

presents more of an opportunity for allocating query units to lightly loaded sites.

A second observation is that RANDOM$_P$ performs better than STATIC in Tests 2 and 3, as partially replicated data causes the system to become unevenly loaded. Randomly picking processing sites helps to equalize the system load somewhat, particularly in Test 3 where each relation has only two copies. In Test 3, the performance of RANDOM$_P$ was found to be close to that of LBQP because it has a 50 percent chance (statistically) of selecting the right site for load balancing in this case. In both tests, a greater improvement in system throughput was observed than in Test 1. In Test 2, the maximum *TIF(LBQP,STATIC)* is 16%, and in Test 3, it reaches 35%. This shows that LBQP effectively

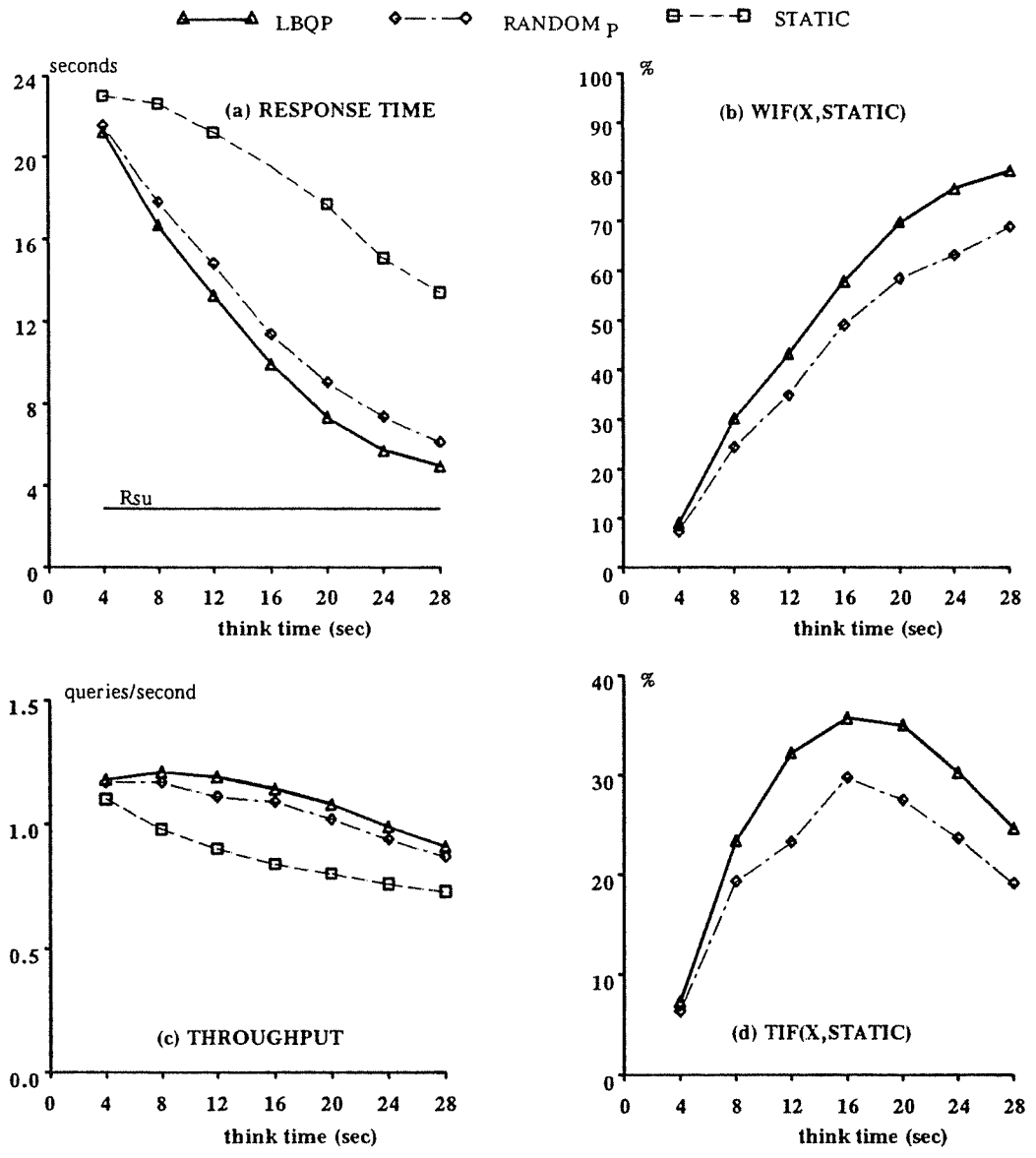

Figure 5: Results of Experiment 1, Test 3 (2 copies/relation).

eliminates resource contention at heavily loaded sites, improving both the response time and the throughput of the system in the partially replicated case.

It can be seen from the parameter settings that the first query unit is likely to be the bottleneck for the queries in these tests, as its service time is two-thirds of the query's total processing time. In Test 4 of Experiment 1, only R_1 (which is referenced by the first query unit) was replicated at every site, and the remaining relations were simply duplicated (Case 4 in Table 7). The results of this test can be summarized briefly as follows: It was again found that LBQP provides a significant improvement in waiting time. It was also found that fully replicating R_1 improves the response time dramatically for both the STATIC and LBQP algorithms — the mean response time for LBQP in this case was nearly that of the fully replicated case.

3.4.2. Experiment 2: Non-Uniformly Loaded Sites

In Experiment 1, every site in the system had the same (statistical) workload. Experiment 2 investigates the behavior of the dynamic query unit allocation algorithm when the sites have different query arrival rates. The workload in this experiment is the same as that of Experiment 1, except that the *mpl* parameter (the number of terminals) was set to 1 at three of the sites (sites $S1$, $S3$, and $S5$) in order to lower their incoming loads and thus obtain non-uniform arrival rates for queries. The relations in this experiment were fully replicated (Case 1 in Table 7).

The results of Experiment 2 are illustrated in Figure 6. The effects of a non-uniform workload on the system's load distribution are similar to those of partially replicated data. RANDOM_F performs a little better here than it did in Test 1 of Experiment 1 (fully replicated data, uniform workload), with slight improvements in its *WIF* and *TIF* being observed when the system is lightly loaded. LBQP performs much better here, as shown by both *WIF*(*LBQP*, *STATIC*) and *TIF*(*LBQP*, *STATIC*). With a *think_time* of 1 second (where the system-wide average CPU utilization is about 0.85), *WIF*(*LBQP*, *STATIC*) exceeds 25%, and *TIF*(*LBQP*, *STATIC*) is greater than 15%. When the system load decreases, the waiting time improvement increases rapidly and the throughput improvement decreases somewhat. With a *think_time* of 8 seconds, the *TIF* is still over 13%, while the *WIF* exceeds 70%. It is clear that LBQP improves the performance of the system significantly when the sites would otherwise be unevenly loaded. Similar results would be expected if the site loads were fairly even but the processing capacities of the sites were different.

3.4.3. Experiment 3: A Mix of Query Types

In Experiments 1 and 2, the workload used in the tests consisted of a single query type. Experiment 3 investigated the performance of LBQP for workloads consisting of a mix of queries with different numbers of query units. 50% of the queries in this experiment had one query unit (referencing R_1), 30% had two (referencing R_1 and R_2), and the remaining 20% had three (referencing all three relations); the query unit parameters for the three query units were otherwise the same as in Experiment 1.

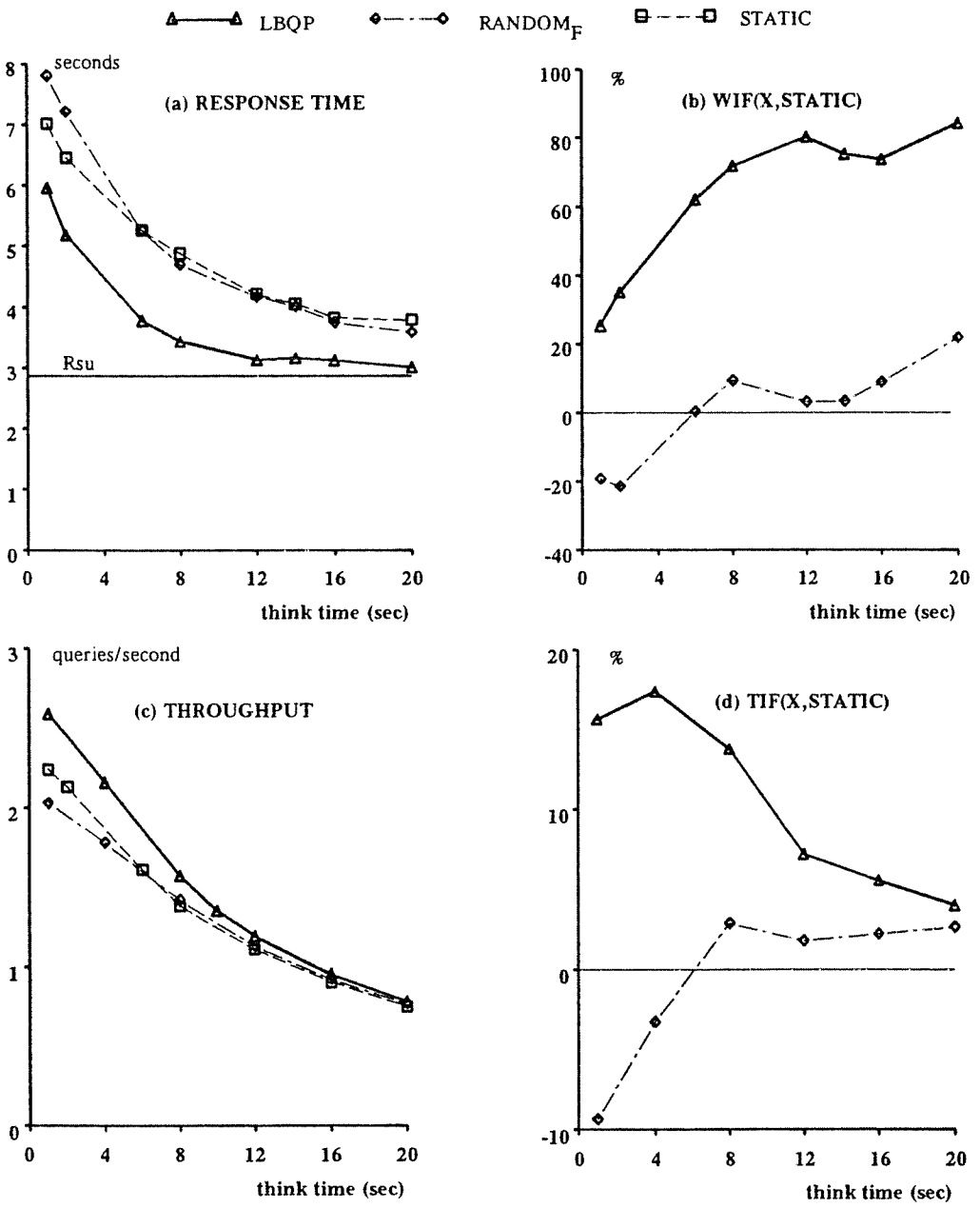

Figure 6: Results of Experiment 2 (non-uniform query arrivals).

Two tests were performed, one with full replication and one with four copies of each relation. The results obtained in this experiment were similar to those of Experiment 1, the main difference being that LBQP outperformed STATIC by more here than it did in Experiment 1. This was because, with all queries wanting to access R_1, 80% wanting R_2, and only 20% wanting R_3, the loads of the sites with the primary copies of the first two relations were higher than those of the other sites here, so load balancing had more of an opportunity to provide gains in this case. Finally, the improvements offered

by LBQP as compared to $RANDOM_p$ were also somewhat greater in this experiment.

4. CONCLUSIONS

This paper has presented a new approach to query processing, an approach that integrates query processing and load balancing for locally distributed database systems. The approach, referred to as LBQP (Load-Balanced Query Processing), uses load information to select the processing site(s) for a query, dynamically choosing from among those sites that have copies of the relations referenced by the query. Query compilation is used to produce a statically-optimized *logical plan* for the query, and then a dynamic optimization phase converts this logical plan into an executable *physical plan* at runtime. This latter phase first allocates subqueries (query units) to sites, then chooses between using a pipelined join-based or semijoin-based algorithm for each of the distributed joins in the resulting plan. A heuristic algorithm was presented for doing load-balanced dynamic query unit allocation, and it was shown that this algorithm has a very high rate of generating the optimal allocation of subqueries to sites, yet it is fast enough to be reasonably used at runtime (unlike an exhaustive approach).

A simulation study was conducted to investigate how a database system might perform using the LBQP approach to distributed query processing. In particular, the study addressed the performance impact of using LBQP's algorithm for dynamically allocating query units to sites. The results of the study indicate that LBQP offers better performance than either static allocation or random allocation strategies, improving the waiting time (i.e., response time) for queries and, in many cases, also improving the throughput of the system. The improvements were found to be most significant with either partially replicated data or non-uniform site loads. Waiting time reductions of 50% or more were typical under moderate CPU loads, and the throughput improvement in some cases reached 10-30%.

This paper represents a first step towards integrating load balancing and query processing, but a number of questions remain. First, the query model used in our performance study was fairly abstract. A truly convincing study would have to either model the behavior of queries at a much lower level or else actually experiment with LBQP in a multi-user distributed database system. Second, the LBQP approach is based on experimentally-gained experience that suggests that static planning can be done on a logically centralized database system, at least when the actual system is based on a local area network. It would be interesting to investigate how well this approach would work for systems based on slower networks. Third, this study has focused completely on read-only queries, ignoring updates. While we do not expect updates to require changes in our algorithms (at least not in systems where transactions are allowed to read any copy of an item but must update all copies), we recognize that updates lead to a performance tradeoff related to the number of copies — the load balancing performance benefits stemming from having multiple copies will trade off against the update costs for the copies, and this tradeoff needs to be examined. Finally, we plan to apply the LBQP approach to more general process structures, and we then plan to examine how the ideas underlying LBQP might be applied to the general problem of load-balancing for distributed programs.

ACKNOWLEDGEMENTS

The authors wish to acknowledge Miron Livny for many helpful discussions, and also for providing us with his DENET simulation tools and advising us on their use. The NSF-sponsored Crystal multicomputer project at the University of Wisconsin provided the many VAX 11/750 CPU-hours that were required for the simulation experiments reported here.

REFERENCES

[Alon84] R. Alonso, "Query optimization in distributed database management systems through load balancing," talk presented at UW-Madison, April 1984.

[Aper83] P. M. G. Apers, A. R. Hevner, and S. B. Yao, "Optimization algorithms for distributed queries," *IEEE Transactions on Software Engineering*, **SE-9**, 1, January 1983.

[Bern81] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve and J. B. Rothnie, "Query processing in a system for distributed databases (SDD-1)," *ACM Transactions on Database Systems*, **6**, 4, December 1981.

[Brya81] R. Bryant and R. Finkel, "A stable distributed scheduling algorithm," *Proceedings of the 2nd International Conference on Distributed Computing Systems*, April 1981.

[Care85] M. J. Carey, M. Livny, and H. Lu, "Dynamic task allocation in a distributed database system," *Proceedings of the 5th International Conference on Distributed Computing Systems," Denver, May 1985.*

[Chiu80] D. M. Chiu and Y. C. Ho, "A methodology for interpreting tree queries into optimal semi-join expressions," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, May 1980.

[DeWi85] D. J. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: design and implementation experience," Computer Sciences Technical Report #553, Computer Sciences Department, University of Wisconsin-Madison, September 1984.

[Eage85] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated dynamic load sharing," *Proceedings of the ACM-SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, August 1985.

[Epst78] R. Epstein, M. Stonebraker, and E. Wong, "Distributed query processing in a relational database system," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, June 1978.

[Hevn79] A. R. Hevner and S. B. Yao, "Query processing in distributed database systems," *IEEE Transactions on Software Engineering*, **SE-5**, 3, May 1979.

[Jark84] M. Jarke and J. Koch, "Query Optimization in Database Systems," *Computing Survey*, **16**, 2, 1984.

[Kamb82] Y. Kambayashi, M. Yoshikawa, and S. Yajima, "Query processing for distributed databases using generalized semi-joins," *Proceedings of the ACM-SIGMOD International Conference on Management of Data," June 1982.*

[Liu82] A. C. Liu and S. K. Chang, "Site selection in distributed query processing," *Proceedings of the Third International Conference on Distributed Computing Systems," Miami/Ft. Lauderdale, Florida, October 1982.*

[Livn82] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," *Proceedings of ACM Computer Network Performance Symposium*, April 1982.

[Livn85] M. Livny, "DENET — a Modula-2 based simulation language," Computer Sciences Department, University of Wisconsin-Madison, in preparation.

[Lohm85] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger, and P. F. Wilms, "Query processing in R^*," in *Query Processing in Database Systems*, W. Kim, D. S. Reiner, and D. S. Batory (editors), Springer-Verlag, Berlin/Heidelberg, 1985.

[Lu85a] H. Lu and M. Carey, "Some experimental results on distributed join algorithms in a local area network," *Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.*

[Lu85b] H. Lu, "Distributed query processing with load balancing in local area networks", Ph.D Dissertation, Computer Sciences Department, University of Wisconsin-Madison, December 1985.

[McCo81] R. McCord, "Sizing and data distribution for a distributed database machine," *Proceedings of the ACM-SIGMOD International Conference on Management of Data," May 1981.*

[Ni82] L. Ni, "A distributed load balancing algorithm for point-to-point local computer networks," *Proceedings of COMPCON*, Fall 1982.

[Seli79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," *Proceedings of the ACM-SIGMOD International Conference on Management of Data," June 1979.*

[Seli80] P. G. Selinger and M. Adiba, "Access path selection in distributed Database management systems," *Proceedings of the First International Conference on Distributed Data Bases*, Aberdeen, 1980.

[Ston85] M. Stonebraker, "The case for shared nothing," presented at the *International Workshop on High Performance Transaction Processing*, Asilomar, September 1985.

[Wang85] Y. T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Transactions on Computers*, **C-34**, 3, March 1985.

[Wong77] E. Wong, "Retrieving dispersed data from SDD-1: a system for distributed database," *Proceedings of the 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, May 1977.

[Yu83] C. T. Yu and C. C. Chang, "On the design of a query processing strategy in a distributed database environment," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, 1983.

[Yu84] C. T. Yu and C. C. Chang, "Distributed query processing," *Computing Surveys*, **16**, 4, December 1984.