Computer Sciences Department

Distributed Query Processing with Load Balancing in Local Area Networks

Hongjun Lu

Technical Report #624

December 1985

UNIVERSITY OF
WISCONSIN
MADISON

DISTRIBUTED QUERY PROCESSING WITH LOAD BALANCING IN LOCAL AREA NETWORKS

by

HONGJUN LU

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1985

ABSTRACT

This thesis presents a new approach to distributed query processing in locally distributed database systems, load-balanced query processing (LBQP), which integrates distributed query processing and load balancing. Several observations about previous research work in distributed query processing motivated this study. First, only a few query processing algorithms have been developed specifically for distributed databases based on local networks. Second, the use of multiple copies of data to improve performance in a distributed database system has not been addressed by most existing algorithms. Finally, and perhaps most importantly, existing query optimization algorithms have considered only the static characteristics of a distributed database. The algorithms reported here consider the dynamic load status of the system and the existence of multiple copies of data to provide better performance than is achievable with purely static planning techniques.

The dynamic query allocation problem for a distributed database system with fully-replicated data was studied first using simulation. Two new heuristic algorithms were proposed for dynamically choosing a processing site for a newly arrived query in a local network environment. Both of these heuristics use knowledge about queries obtained during query optimization, such as their estimated CPU and I/O requirements. A simulation model was developed and used to study these heuristics and compare them to an algorithm that simply balances the number of jobs at each site. The results of this study indicate that knowledge of query processing requirements can be used effectively to improve

ACKNOWLEDGMENTS

I am greatly indebted to my advisor, Professor Mike Carey, for his extreme earnestness, excellent guidance, continued encouragement, and generous support, as well as extraordinary patience in improving my writing skills. I would also like to thank the readers of this thesis: Professor Miron Livny, who introduced me to the distributed system performance area and provided the simulation languages used in this thesis, and Professor David DeWitt, who was the source of many constructive ideas. All three readers of my thesis were extremely generous with their time in helping me to finish. I would also like to thank Professors Mary Vernon and Charles Kime for making the time and having the kindness to serve as members of my exam committee, and for providing helpful comments.

Special thanks must be given to the primary investigators and staff of the Crystal project for providing an excellent research environment. Tad Lebeck, Tom Virgilio, Nancy Hall and Robert Gerber were always willing to help with questions and problems. Thanks are also due to the members of WiSS Project. In particular, Hong-Tai Chou provided me with the instrumented version of WiSS and helpful suggestions. I would also like to thank the staff of the IBM-WISC Net Project for providing a large number of CPU hours for some of my simulation experiments.

I am grateful to Professor Ed Desautels and Paul Beebs, and to all my friends and fellow students in the CS department's Systems Laboratory. They made my work there memorable.

overall system performance through dynamic query allocation.

In order to obtain empirical results regarding distributed query processing in local area networks, a testbed was built using an experimental distributed system, the Crystal multicomputer, to conduct experiments on the performance of distributed join algorithms. Eight different distributed join methods were implemented using the testbed. Join queries with a variety of relation sizes, join selectivities, and join column value distributions were experimentally studied. The performance results obtained indicate that pipelined join methods outperform sequential methods over a wide range of join queries. It was also found that the communications costs in a local network environment are certainly not a dominant factor with respect to performance.

A three-phase load-balanced query processing algorithm, Algorithm LBQP, was developed based on these experimental results and the results of the study of dynamic query allocation. This algorithm first statically generates a processing plan for a query in a locally distributed database system, ignoring the physical storage sites of the relations referenced by the query. A dynamic query unit allocation algorithm is then applied to the plan to determine the processing sites for each relation. Finally, specific processing methods for the distributed joins in the resulting plan are selected. A model of distributed database systems with partially-replicated data was used to investigate the performance of the dynamic query unit algorithm of LBQP. The results show significant improvements in performance, including improvements in both the mean waiting time for queries and the overall system throughput.

I am deeply grateful to my parents, my brothers, my sisters and my in-laws for their support and encouragement. To my wife, Juqing, and my daughter, no words can express my deepest gratitude. They deserve more than mere thanks for their years of sacrifice and their endurance of hardships both material and psychological.

This research was supported by National Science Foundation Grant Numbers DCR-8402818 and MCS-8105904, an IBM Faculty Development Award and the Wisconsin Alumni Research Foundation.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Distributed computing systems have been a most active research area in computer sciences for the last decade. In the database management systems (DBMS) area, more and more attention has been paid to the new problems posed by the distribution of a database among different machines. Query optimization in a distributed environment, one of the important research topics in the distributed DBMS (DDBMS) area, has been examined by a large number of researchers [Aper83] [Bern81c] [Blac82] [Chan82a] [Chiu80][DBE82] [Epst78] [Epst80a] [Good79] [Goud81] [Hevn80] [Kamb82] [Kers82] [Lohm85] [Seli80] [Wong77] [Wong80] [YuCh83] [Yu85]. In the operating systems area, an important topic related to distributed computing has been task allocation and load balancing. Researchers in this area attempt to improve the performance of a distributed system by distributing tasks properly among the available resources [Bokh79] [Chu80] [Chow79] [ElDe78] [Gyly76] [Livn83] [Ma82] [Ni81] [Pric79] [Rao79] [Ston77b] [Ston77a] [Ston78].

Despite the large amount of work that has been devoted to these topics, little has been done to merge results from these two different areas to explore their combined potential for performance improvement in distributed database systems. The work presented in this thesis aims to improve the performance of distributed database systems by integrating dynamic load balancing and distributed query processing algorithms. In this introductory chapter, previous research work in the areas of distributed query processing and load balancing is reviewed.

The motivation for this research and an overview of the thesis are given at the end of the chapter.

1.1. DISTRIBUTED QUERY PROCESSING

Many distributed query processing algorithms have been proposed in the literature [Wong77] [Epst80a] [Bern81c] [Hevn79] [Good79] [Wong80] [Seli80] [Aper83] [YuCh83]. Detailed reviews of distributed query processing can be found in a number of survey papers [Sacc82] [YuCh84]. Some issues and results from previous work that is closely related to this research are briefly discussed here.

1.1.1. The Objective and Cost Functions

The *objective function* of a query optimization algorithm quantitatively specifies the main goal for optimization. The goal can be to minimize the total processing cost for a query (including I/O cost, CPU cost, and communications cost), or to minimize its response time. Minimizing communications cost is also an objective of a number of optimization algorithms.

The *cost function* is used by a query optimizer to estimate the cost of the steps of a processing plan. The cost of processing a distributed query, C, consists of two basic components, the local processing cost C_{local} and the communications cost C_{comm}.

$$C = C_{local} + C_{comm}$$

Local processing costs are usually decomposed into the CPU cost C_{CPU} and the I/O (disk) cost C_{IO}, while the communications cost can be represented as the sum of a fixed message setup time C_0 and a variable cost $C_m(X)$ that is propor-

tional to the amount of data to be transferred, X.

$$C_{local} = C_{CPU} + C_{IO}$$

$$C_{comm} = C_0 + C_m(X)$$

Not every component of the processing cost is considered by all of the proposed algorithms, however. Table 1.1 shows the cost components considered by a number of different distributed query optimization algorithms. The symbol "X" indicates that the component is included in the cost function of the algorithm. It can be seen that the access path selector of System R^* is the only algorithm which takes all components of the cost into consideration. The CPU cost, I/O cost and message cost are all weighted and combined together into the overall processing cost in system R^*.

1.1.2. The Evaluation Function and Search Heuristics

One feature of distributed query optimization is its tremendous search space. The size of this search space is determined by the physical database

The Cost Components of Different Algorithms				
Algorithm	Local Cost		Network Cost	
	C_{CPU}	C_{IO}	C_0	$C_m(X)$
SDD-1,OPT [Bern81c]				X
Distributed INGRES [Epst78]	X	X		X
System R^* [Seli80]	X	X	X	X
Apers-Hevner-Yao [Aper83]		X	X	
Yu-Chang [YuCh83]			X	X

Table 1.1: The cost components of different algorithms.

organization and the number of relations and attributes referenced by the query. Although experiments [Epst80b] indicated that there is a dramatic difference between the quality of plans produced by limited and exhaustive search, and that exhaustive search performs consistently better, most algorithms still use a limited search strategy. One of the few algorithms that uses an exhaustive search is the access path selector of System R^* [Seli80]. Other algorithms, including Wong's algorithm, SDD-1's Algorithm OPT, Apers-Hevner-Yao's algorithm, and the Distributed INGRES algorithm, use a hill-climbing search method. The basic feature of this method is that the possible processing steps are evaluated one by one, without looking ahead or backwards. The locally optimal step is always chosen as the next step. As a result, the global optimality of the result is not guaranteed, as a globally optimal processing strategy will not be found unless it is locally optimal at each step as well.

When comparing alternative processing steps, a metric is required for comparison purposes. This is specified by the *evaluation function*. Most algorithms use *processing cost* as the evaluation function. At decision making time, the costs of different alternatives are compared and the one with the least cost is viewed as the optimal one. In the algorithms with the objective function of minimizing the communications cost, the processing cost is often represented as the amount of data transferred among different sites (in bytes). In such algorithms, two other quantities, *benefit* and *profit*, are also sometimes used in evaluating processing plans. The benefit of a processing step is its reduction effect, denoted by the size change (in bytes) of the result relation. The *profit* of a processing step is the difference between its cost and its benefit. Algorithm OPT of SDD-1 uses profit as its evaluation function [Bern81c], and the most

profitable processing steps are selected as the final plan. Black proposed a heuristic for a processing algorithm similar to OPT [Blac82]. The least cost and maximum benefit criteria are both used during the search. Since these two criteria are usually not satisfied simultaneously, the algorithm first finds the processing steps with the least cost, and then chooses the one with the most benefit. That is, there is a trade off made between these two extremes. Black's simulation results indicated that this heuristic is much better than that of both Hevner's algorithm and the SDD-1 algorithms in the sense that the resulting plans are closer to the optimal ones.

1.1.3. The Semijoin and Join Operators

The semijoin was proposed as a useful primitive operator to reduce the data transfer cost in distributed database systems. The semijoin $R_a < A = B] R_b$ is defined as the join $R_a [A = B] R_b$ projected on the attributes of R_b. This operator has several properties of interest:

(1) A semijoin always reduces the size of the database. In contrast, the join operator can lead to a Cartesian product in the worst case.

(2) The communications cost incurred in a semijoin is usually less than for the join operation since only the join attribute values and the results of the semijoin need to be transferred.

(3) There is a class of queries that can be processed completely using a sequence of semijoin operations [Bern79b]. Furthermore, a join operation can always be replaced by two semijoins.

A number of distributed query processing algorithms have been developed based on the semijoin operation [Bern81a] [Bern81c] [Goud81] [Kamb82] [YuCh83]

[Yu85].

However, there are several algorithms that do not use semijoins at all. Distributed INGRES and System R^* are examples [Sch80]. A related method known as "fetch inner tuples as needed" is used as one alternative to process distributed joins in System R^*. This method is similar to the semijoin, but it is not exactly the same. Using this method, the join R_a [A=B] R_b is processed as shown in Figure 1.1. It can be seen that relation R_a is scanned just once during the join. The tuple of R_a that is currently being processed stays in memory to wait for the arrival of matching inner tuples. This is different from the semijoin method, where relation R_a is scanned more than once: once to complete the

---

Site of R_a :
  Repeat
    Fetch a tuple t_a from R_a;
    Send $t_a.A$ to site of R_b;
    Receive t_b tuples from R_b;
    Join t_b's with t_a;
  Until all tuples in R_a have been processed.

Site of R_b :
  Repeat
    Receive a tuple t_a from R_a;
    Find all t_b where t_b.B = t_a.A;
    Send these t_b's to site of R_a;
  Until no more tuples are received.

Figure 1.1: The "fetch inner tuples as needed" method of System R^*.

---

projection of R_a on its join attributes, and again to perform the join of R_a with the matching tuples returned from R_b. Another difference between these two methods is that, in the "fetch inner tuples as needed" method, the join attributes are sent one by one. In the semijoin, the join attributes of all tuples are projected out first and sent together (with duplicate values eliminated).

1.1.4. The Query Allocation Problem

Data replication is one way to obtain high data availability as well as reliability in distributed database systems [Alsb78] [Hamm80]. In a database with replication, some data items are stored redundantly at multiple sites. If a user query references such a data item, one of its sites has to be selected as the processing site. This problem is similar to the task allocation or resource allocation problem in operating systems, and is referred to as the *query allocation problem* here.

Most existing algorithms have ignored the query allocation problem 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 [Bern81c][Aper83][Epst78][Hevn79]. A good query execution plan is then selected given this materialization. The designers of System R^* 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 [Seli80].

Algorithms that consider site selection during query optimization were proposed by Liu and Chang [Liu82]. In their algorithms, a user query that has been translated into a sequence of relational algebra operations is represented by

a transaction graph. This graph specifies the order of the operations, the relations referenced by each operation, and the sites where the relation is stored. The processing costs of the query are then estimated by traversing the graph and computing the local processing costs and the communications costs when different copies of the referenced relation are used. The sites that lead to the minimum cost are chosen as the processing sites. Their algorithms can be applied to a subset of the class of all queries — queries whose transaction graphs are trees.

Yu and Chang proposed a copy identification algorithm that chooses the processing sites for a given query and data distribution [YuCh83][Yu85]. Since the copy selection problem is NP-hard, even for simple queries, their algorithm uses some heuristics. When communications cost is dominant as compared to local processing cost, the algorithm chooses the minimum set of sites containing all relations referenced by the query as the processing sites. If local processing cost is significant, they suggest selecting maximum set of sites containing the relations referenced by the query in order to increase parallelism. This copy identification algorithm is applied before the selection of the optimal processing plan is performed. (In their case, the processing plan is a semijoin sequence).

1.1.5. Resource Contention

All distributed query processing algorithms to date have been based on the static characteristics of a distributed database system. Both the cost functions and the evaluation functions ignore dynamic characteristics of the system such as transaction queuing effects caused by heavily loaded processors or the network. However, dynamic system characteristics can be important factors which

affect the optimality of a processing plan. Take an extreme case as an example, where a number of users at a site submit the same (or very similar) queries during a short period of time. If the load situation of the system is not considered, most query processing algorithms will generate the same processing plans for all of the queries. These plans will have the same resource demands, and hence will compete with each other. Some transactions will thus have to wait in long queues for highly utilized resources at some of the sites, while other sites with the same processing capabilities may be idle. Simulation results reported by McCord indicate that significant performance degradation occurs when long queues form at some of the sites in a distributed database system [McCo81].

1.2. TASK ALLOCATION AND LOAD BALANCING IN DISTRIBUTED SYSTEMS

Task allocation and load balancing have been a major issue associated with distributed computing systems. As Livny and Melman showed, the probability that at least one processor is idle while tasks are waiting at other sites in a distributed system (a "wait while idle" state) is remarkably high over a wide range of network sizes and processor utilizations [Livn82]. Load balancing aims to reduce this probability through redistributing the load of the system resources. As a result, the queuing times of tasks are reduced and better system performance can be obtained.

In order to achieve a load-balanced system, tasks submitted to a distributed system should be allocated to the sites in the system properly. This is known as the *task allocation* problem. However, a good initial assignment does not necessarily lead to a balanced system afterwards. *Task migration* is another pos-

sible step for improving the system's performance. Task migration is the process of transferring partially executed tasks at one site to an idle or less loaded site to continue their execution. Task allocation problems can be solved either statically or dynamically, while task migration is dynamic by nature. Static task allocation algorithms consider a fixed set of tasks, and they use only information about the static properties of a system, such as the number of processors, the processing capability of each processor, the memory size of each processor, etc. and/or aprior information on tasks. Allocation plans generated by static allocation algorithms are therefore independent of the current system state. Dynamic allocation, on the other hand, employs information about the current system state in making allocation decisions. The remainder of this section briefly reviews some research issues and results on load balancing that are closely related to the problem studied in this thesis.

1.2.1. The Static Task Allocation Problem and Its Solution Methods

The basic task allocation problem is to assign tasks to the sites in a system whose capabilities are most appropriate for the tasks such that some objective is achieved. The tasks can be either different processes from one user's program, or they can be different programs from different users. These tasks may communicate among one another. The objectives of most proposed task allocation algorithms are to minimize the interprocess or intermodule communication costs (IPC or IMC) and/or to minimize the completion time of a number of tasks. In some cases, there are additional constraints such as the memory space required by the tasks. A number of solution methods have been developed for the task allocation problem, including *graph theoretic methods, integer programming methods, brand-and bound search methods* and *heuristic methods.*

Graphic theoretic methods. The allocation of program modules in dual-processing systems, as studied by Stone [Ston77b], is a typical example of such methods. The modules to be assigned and the two processors themselves are represented by the nodes in a graph. The arcs in the graph are weighted to represent the intermodule communications cost. Network flow algorithms are then applied to this graph to find the maximum flow minimum cutset that corresponds to the allocation plan with the minimum total processing cost (the sum of module execution costs and intermodule communications costs). This method was later extended by the author and other researchers to add additional constraints [Ston78] [Ston77a] [Rao79] [Bokh79].

Integer programming methods. Integer programming techniques can be used to solve the task allocation problem if the problem can be formulated as an optimization problem. For example, a task allocation problem of assigning m tasks to a system with n processing sites can be viewed as a problem with $m \cdot n$ variables. Let:

(1) $X_{ij} = 1$ denote the fact that task i is assigned to site j (and 0 otherwise);

(2) E_{ij} be the execution cost of task i if it is assigned to site j; and

(3) the communication cost between two tasks i and k that are not assigned to the same site be C_{ik} (and 0 otherwise).

The objective function of minimizing the total cost can then be expressed as

$$\sum_{i=1}^{m}\sum_{j=1}^{n} E_{ij} X_{ij} + \sum_{i=1}^{m-1}\sum_{k=i+1}^{m} C_{ik} - \sum_{i=1}^{m-1}\sum_{j=1}^{n}\sum_{k=i+1}^{m} C_{ik} X_{ij} X_{kj}$$

The first term is the sum of the execution costs of all tasks, the second term is the total communications cost if all tasks are allocated to different sites, and the

third term represents the communications cost eliminated due to pairs of tasks that are allocated to the same site. Since $\sum_{i=1}^{m} \sum_{k=i+1}^{m} C_{ik}$ is a constant, the objective function can be more simply expressed as:

$$\sum_{i=1}^{m} \sum_{j=1}^{n} E_{ij} X_{ij} - \sum_{i=1}^{m-1} \sum_{j=1}^{n} \sum_{k=i+1}^{m} C_{ik} X_{ij} X_{kj} \qquad (1.2.1)$$

The constraint that each task can only be assigned to one processor can be expressed as:

$$\sum_{j=1}^{n} X_{ij} = 1 \qquad \text{for} \quad 1 \le i \le m \qquad (1.2.2)$$

This is a 0-1 integer programming problem [Pric84], so task allocation can be performed by solving equation (1.2.1) subject to the constraint of (1.2.2). A number of algorithms based on such methods have been proposed [Gyly76] [Pric84].

Bound-and-branch search methods. The task allocation problem can also be viewed as a search problem. An algorithm that uses a branch-and-bound search method was proposed by Ma, Lee and Tsuchiya [Ma82]. In their algorithm, the tasks to be allocated are considered one by one. For every possible assignment of a task, the total processing cost is calculated up to the current assignment. If this cost is greater than the current bound, the assignment is pruned to reduce the search effort. The optimal plan is found in this way after all tasks are assigned. Other objectives and constraints of the problem can be included as input to the algorithm and considered during the assignment. For example, in order to prevent some tasks from being assigned to the same processors, a "task exclusion" matrix can be included as input to the algorithm. Also, a "task preference" matrix can be used to indicate the suitability of a task for a

processor.

Heuristic search methods. Heuristics are widely used in search problems to reduce the search effort and still obtain optimal or nearly optimal solutions. Several algorithms have been proposed which apply heuristic search techniques to the task allocation problem [Gyly76] [Chu80] [Pric84]. One simple and effective heuristic that is sometimes used is that the two modules with the greatest intermodule communication cost should be assigned to the same site first.

1.2.2. Dynamic Task Allocation and Task Migration

In a distributed computing system, the probability that at least one processor is idle while tasks are waiting at other sites, P_{wi}, is remarkably high over a wide range of network sizes and processor utilizations [Livn82]. Eager, Lazowska, and Zahorjan recently studied the potential benefit of dynamic load balancing using an analytical model [Eage84]. Their results indicate that even a very simple dynamic load balancing mechanism can provide dramatic performance improvements. However, it is not easy to implement a dynamic load balancing algorithm. The representation and estimation of load, the load information exchange policy and the control policy are all major concerns in designing and implementing load balancing algorithms.

1.2.2.1. Load Representation and Estimation

The proper representation and accurate estimation of load information is a key problem in load balancing algorithms. The basic requirement for such representation and estimation is that it must be simple, effective, and efficient. The representation of load information should be simple enough so that the

inter-computer exchange of load information introduces as little additional overhead to the system as possible. To be effective and efficient, the estimation must be fairly accurate and able to be evaluated in a short time period in order to reflect the current status of the system.

In a number of research efforts, the load of a processing site is defined as its degree of "busyness". According to this view, the load of a site S_i, $L(S_i)$, is represented by the number of tasks being currently served at that site, $N(S_i)$ [Ni81a] [Livn83]. That is,

$$L(S_i) = N(S_i)$$

Under this definition, site S_i is idle when $L(S_i) = 0$. When $L(S_i) > 1$, there are tasks waiting to be executed at site S_i. (In their models, a site consists of a single server.)

A different approach is taken in the algorithm proposed by Bryant and Finkel [Brya81]. Their algorithm views the load of a site as its ability to complete new tasks. Under this view, the load of a site S_i with respect to task k, $L(S_i,k)$, is the estimated *response time* of task k if it were to be processed at S. That is,

$$L(S_i,k) = RES(S_i,k)$$

This view seems more attractive for situations where minimizing response time is the main objective. An algorithm to estimate this response time $RES(S_i,k)$ was also proposed in the paper. Their response time estimate is

$$RES(S_i,k) = \sum_{j \in S_i} \min(R_E(j),R_E(k)) + R_E(k)$$

where $R_E(j)$ is the remaining service time of job j.

Ferrari recently proposed another alternative load measurement [Ferr85]. In this work, a site is modeled by a closed multichain queuing network with service centers that have population-independent service rates and are scheduled by different scheduling policies. The mean value analysis method [Reis80] is applied to the model to derive the load measurement, which is the increase in task response time because there is more than one task served by the site. This proposed measurement is a linear combination of the mean queue lengths at the site being considered, the coefficients being the total times the task would spend in each service center if it ran alone at the site.

1.2.2.2. Information Policy

The information policy of a load balancing algorithm specifies how the load information described above is stored, collected, and exchanged. Since the exchange of load information increases network traffic, and storing and collecting load information introduces overhead at each site, a good information policy is an important component of a load balancing algorithm.

Load information could be stored at one site in the system, and other sites could then access this information whenever needed. The advantage of such a global approach is that it is simple to implement and provides system-wide load information. However, this is only practical in local networks of small sizes or certain topologies, as keeping all information at one site is both expensive and unreliable. The most common approach is that each site in the system keeps a *load vector* to store load information about the other sites. Decision making is then based on this locally stored information. Thus, the problem becomes one of how to keep an up-to-date load vector with a reasonable amount of overhead.

The solution is to select a *balancing region* of the proper size and an appropriate information exchange scheme [Livn83].

The *balancing region* of a site S_i is a subset of a distributed system that consists of those sites which S_i will consider as candidates for receiving one or more of its tasks. A larger balancing region leads to a better global balancing effect, with the disadvantage being higher message traffic if broadcast communication is not available. The choice of a balancing region in systems without a broadcast mechanism is quite critical [Livn83]. For example, if the balancing region extends too far in a point-to-point network, the transmission time for returning the results of a remotely executed task will be difficult to predict since the load of the communication channel can change frequently. Furthermore, the routing algorithm has to be invoked to find the best path to send the result back, introducing more overhead. The usual choice is to include only the directly connected processors in the balancing region for a processor in a point-to-point network.

For exchanging load information within a balancing region, a number of schemes have been proposed. One scheme is to have each site broadcast every change in its load status — whenever a task arrives at or departs from a site, the site broadcasts its new status to the other sites in its balancing region. In this scheme, each site has the most recent information about all other sites, and therefore the task migration decision can be based on accurate information. The drawback of this strategy is its large overhead. One way to reduce this overhead is to have sites broadcast their load status periodically [Livn83]. If the broadcast interval is chosen appropriately, the use of out-of-date load information can still be avoided.

If a system migrates tasks only when some site in the system becomes idle, the only information needed is an "idle" message that is sent whenever a site enters an idle state. With this approach, message cost is minimized, and much of the cost is incurred at an otherwise idle site. The Broadcast When Idle (BID) and Poll When Idle (PID) algorithms [Livn83] are two examples of this approach. The information policy used in the distributed load balancing algorithm proposed by Ni [Ni82] can be viewed as a variation of this scheme. In Ni's algorithm, a status message is broadcast to all neighbors when a site enters an idle state from a busy state or a busy state from an idle state. As long as a site is busy, it will not report the change in the number of jobs in its queue.

Another example of a different sort of information exchange policy is the "exchange while pairing" scheme of Bryant and Finkel [Brya81]. Before a task migration decision is made, a site that wishes to send out tasks asks each of its neighbors in turn to establish a "pair" for possible job migration. Load information from the originating site is included in the request. The pairing process is terminated whenever a mate is found, and load information exchange is also stopped.

1.2.2.3. Control Policy

The control policy of a load balancing algorithm determines the origin, the destination and the time at which to migrate a task. It also determines which site(s) will initiate the load balancing algorithm and the number of tasks to be migrated. As discussed above, A task transfer can be initiated when one site in the system becomes idle. That is, when a site becomes idle, it checks load information kept locally or it sends request messages to its neighbors. A heavily

loaded site is then chosen as the source for task migration. However, it is not necessary to postpone the transfer tasks until the system enters a "wait while idle" state. Tasks can be transferred earlier to prevent the occurrence of such a state, and better system performance can be achieved [Livn83].

The initiator of a task transfer can be either a heavily loaded site that asks for help (the task sender), or a lightly loaded site that is willing to help (the task receiver). The main disadvantage of having a heavily loaded site invoke a transfer is that this further increases the burden on the already heavily loaded site. Also, without proper coordination, more than one heavily loaded site may migrate its jobs to the same lightly loaded site and in turn make it overloaded. Thus, many algorithms allow a lightly loaded site to invoke the transfer instead. In Livny's BID and PID algorithms [Livn83], a site that enters an idle state sends messages to inform its partners that it is ready to receive a job. In the algorithm proposed by Ni [Ni82], a processor invokes the balancing algorithm when the scheduling queue of a processor becomes empty. Wang and Morris reported results indicating that, with the same level of information available, receiver-initiated algorithms have the potential for outperforming sender-initiated algorithms [Wang85]. This conclusion was derived for cases where interprocess communication overhead was negligible. Recent results reported by Eager, Lazowska, and Zahorjan [Eage85] comparing receiver-initiated and sender-initiated load balancing strategies indicate that neither of the strategies dominates the other for all system characteristics and parameter values. If the system load is light to moderate, or if the cost of transferring executing tasks is significantly greater than the cost of transferring newly arrived tasks, then sender-initiated policies are recommended. If the system load is high and the transfer costs are

comparable for the two strategies, then receiver-initiated policies are recommended.

When the balancing algorithm is invoked, task migration may occur. There are several common criteria regarding task migration. The most important is that the migration should be fruitful and stable. Processor thrashing, a situation where a task continually migrates around the network without accomplishing any useful work, should be avoided [Brya81].

1.3. MOTIVATIONS AND THESIS OVERVIEW

In the last two sections, various aspects of distributed query processing and load balancing issues identified by research work in both the database management systems field and the operating systems field were briefly reviewed. One basic observation is that the existing distributed query processing algorithms are still based on static system characteristics, while load balancing is known to be able to improve the performance of distributed computing systems. If queries in a distributed database system can be processed in such a way that the loads of different sites in a database system are equalized to some extent, resource contention will be relaxed and better overall system performance can be expected. The possibility for doing so is provided by multiple copies of data which are to be likely available in a locally distributed database system.

The job of balancing the load in a distributed database system could be charged to the operating system which supports the DBMS. However, there is another alternative: Let the DBMS take the responsibility. In this latter approach, the load balancing mechanism becomes an integral part of the query optimizer. The query processing plans chosen are those that will lead to a bal-

anced system. This alternative has several attractive features:

(1) The query optimizer produces important information about a query, such as estimates of its resource demands (including its I/O and CPU requirements). This information may only be approximate, but it can be used to guide the allocation of queries to sites.

(2) A query is usually decomposed into subqueries during its execution. The query processing problem thus has clear boundaries between steps, providing natural decision-making points for load balancing. For example, subqueries could be the unit of allocation and migration. (Not all ordinary computing tasks are so structured).

(3) To have the DBMS migrate a query is easier and more natural. The following steps are need to migrate a process at the operating systems level : (i) remove the process from execution, (ii) ask the source kernel to move the process, (iii) allocate a process state on the destination processor, (iv) transfer the process state, (v) transfer the program, (vi) forward pending messages, (vii) clean-up the process state and (viii) restart the process [Powe83]. If migration is instead initiated by the DBMS at the right point, some of these steps can be omitted — migration will need only to initiate a new process at the destination site. There is no need to save or clean-up process states. Also, no message forwarding is needed.

The potential benefits of load balancing in distributed database systems combined with the advantages of incorporating the load balancing mechanism into the DBMS motivated the research work described in this thesis. The remainder of this thesis is organized as follows:

Chapter 2 demonstrates the potential of dynamic query allocation in fully replicated distributed database systems. Three basic algorithms for dynamic query allocation in such systems are proposed. A closed, two-class queuing network model is established for a fully replicated distributed database system, and a simulation study based on this model is presented. The results indicate that dynamic query allocation is a promising way to improve system performance, and that the queuing time of queries can be significantly decreased in this way.

Chapter 3 describes an experimental study of the performance of different distributed join algorithms. The algorithms tested, the testbed that was built and the results that were obtained are presented. This study shows that local processing cost is the dominant part of the total query processing cost for local networks. Pipelined join processing was found to yield better performance than sequential join methods in most cases. This study also leads to observations that are important for the design of a load-balanced query processing algorithm (which was the motivation for this study in the first place).

Chapter 4 presents the design and details of a load-balanced approach to distributed query processing. Possible ways to achieve load-balanced query processing are discussed first. Algorithm LBQP, a three-phase load-balanced query processing scheme, is then described in detail. The rationale behind the heuristics used in the first phase, the static planning phase, are analyzed. The main part of the algorithm is the second phase — dynamic allocation. The optimality and complexity of an algorithm for this phase is studied. The results indicate that the proposed heuristic algorithm generates the same optimal plans obtained by exhaustive search in most cases, and in much less time. Finally, the purpose and methods used for the last phase, the refining phase, are discussed.

In order to indicate the effectiveness of the proposed load-balanced approach to query processing, a queuing network model for partially-replicated distributed database systems was established, and simulation experiments were carried out using this model. Chapter 5 describes the model and the results of these experiments. The results indicate that load-balanced query allocation significantly decreases the mean waiting time of queries and increases system throughput in systems with multiple copies of data.

Finally, Chapter 6 summarizes what was learned from this study and indicates directions for future research.

CHAPTER 2

DYNAMIC QUERY ALLOCATION: THE FULLY REPLICATED CASE

In order to investigate the potential of integrating a load balancing mechanism with distributed query processing and to explore possible query placement criteria, the problem of dynamically assigning queries to sites in a distributed database system with fully replicated data is studied in this chapter. Several heuristic algorithms are proposed for achieving good query allocations dynamically. A simulation model is developed for studying these algorithms, and the results of a set of simulation experiments using this model are presented and discussed.

2.1. DISTRIBUTED DATABASE SYSTEMS WITH FULL REPLICATION

A database system is said to be fully replicated if every site in the system has a copy of all relations in the database. User queries can always be processed locally (i.e., processed with no data movement) in this case. This case is easier to deal with from a load balancing prospective than the general case, as the only communications costs which need to be considered in query allocation are the costs of initiating a query at a remote site and then getting the results back to the originating site. Thus, the fully replicated case was selected for the purpose of this feasibility study.

The problem of load balancing in a fully replicated distributed database system might also have practical applications by itself. Consider a large information center that consists of a number of computers with large, high speed storage systems connected via a communications subnet. Requests for information

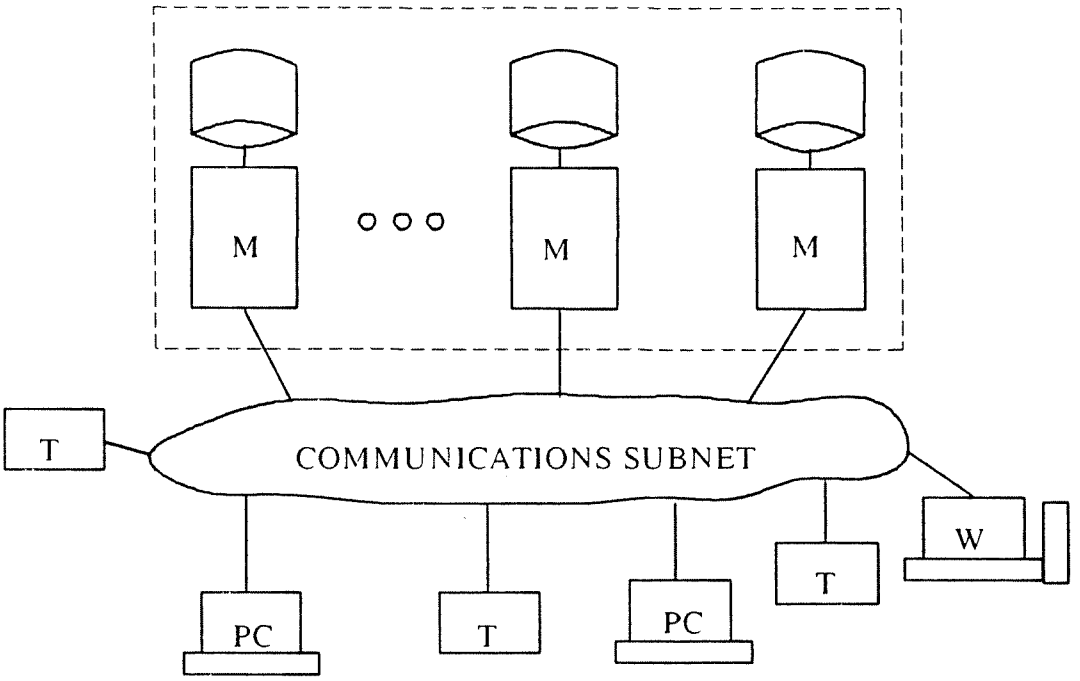

Figure 2.1: A large information service center.

retrieval might come from end user terminals (T), personal computers (PC), workstations (W) and other computers which are connected to the same communications subnet, as shown in Figure 2.1. In order to provide fast and reliable service, the information stored in the database could be fully replicated among these computers.[†] The problem of load balancing among the computers in such a system is exactly the query allocation problem being studied in this chapter.

[†] Such information centers mainly provide information retrieval services to customers, so having to update multiple copies may not be a serious problem.

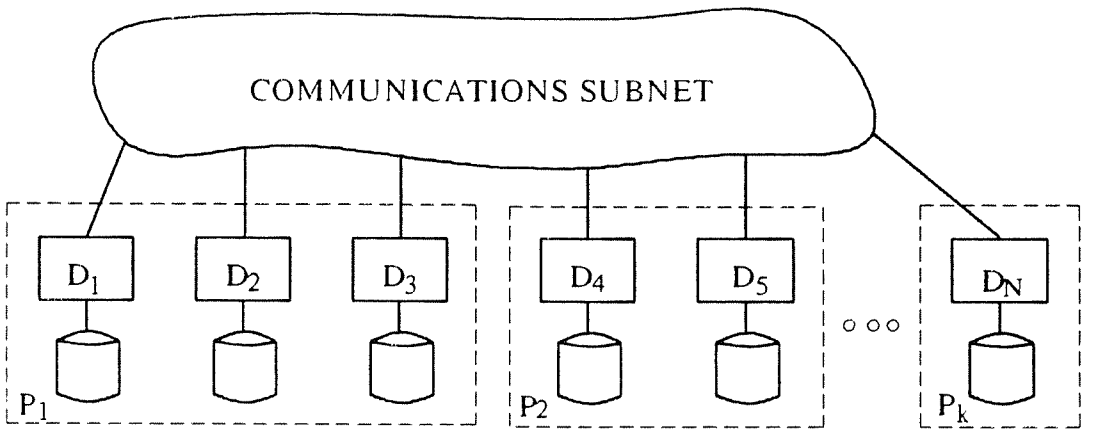

Figure 2.2: A partitioned distributed database system.

The dynamic query allocation problem for the fully replicated case can also be thought of as a basic problem underlying more general load-balanced query processing. As as example, consider the system shown in Figure 2.2. This system consists of N database servers D_i, $1 \leq i \leq N$. The database is designed in such a way that it is divided into P partitions with no data overlap, and each partition of the database is replicated on several machines so as to ensure availability. A query processing plan in such a system consists of subqueries, Q_k, interwoven with intersite data moves, and each subquery will be executed at a site within some partition P_l. Within the partition, then, the data referenced by Q_k is fully replicated. The dynamic allocation problem for subquery Q_k in partition P_l is the same problem as the dynamic allocation problem for a complete query in the fully replicated case.

2.2. HEURISTICS FOR DYNAMIC QUERY ALLOCATION

In this section, three heuristic algorithms are presented for query allocation in the fully replicated case. Each of these algorithms allocates a newly arriving query to a site in the system where it will be processed to completion. The goal of each of these algorithms is to dynamically achieve a load-balanced system which performs better than a system without load balancing. The basic heuristic used by the algorithms is to choose the processing site with the minimum estimated cost. Figure 2.3 describes the selection process in this manner. (One detail not shown in the figure is that the "foreach" loop that examines possible

```
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 2.3: Function to select processing site for query.

remote execution sites should scan these sites in a round-robin fashion). In the case that the processing cost of the local site is the minimum, the local site is always chosen as the processing site. What is different from algorithm to algorithm is their cost estimation procedures. Before describing the details of these cost estimation procedures, two assumptions are made.

(1) The query allocation algorithms have information about the estimated processing costs of the newly arrived query. This processing cost is represented by a triple {*num_reads, page_cpu_time, result_fraction* }, and this cost information is assumed to have been produced by the query optimizer. *Num_reads* is the number of disk accesses needed for retrieve the data from the database. *Page_cpu_time* is the CPU time needed to process a page that has been fetched into the memory. *Result_fraction* is the ratio of the number of result pages and *num_reads*. This last component is used to estimate the communications cost for sending the result of a query back to its originating site if it is executed remotely.

(2) Every site in the system has knowledge about the load status of other sites needed by the query allocation algorithms.

Each of the three query allocation algorithms are now considered in turn.

2.2.1. Balance the Number of Queries

One simple heuristic for dynamic query allocation is to try to keep the *number* of queries at each site evenly balanced. This approach is called the BNQ ("Balance the Number of Queries") algorithm. Each site knows the current distribution of queries in 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 2.4 gives the cost estimation function for the BNQ heuristic. This balancing goal is similar to the approach that several previous load balancing algorithms have used [Livn82] [Livn83] [Ni81] [Ni82] and is fairly typical of the approach taken at the operating system level. It does not use any information about the arriving query, so it is referred to as a non-information-based query allocation algorithm. Its knowledge about the load status of a site is also very simple — it uses only the number of queries at the site.

2.2.2. Balance the Number of Queries by Resource Demands

The 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. (One could perhaps generalize this classification along the lines of the work of Ferrari [Ferr85].) When a new query is initiated from a terminal at some site, this query is classified as being either an I/O-bound query or a

---

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

Figure 2.4: Cost estimation function for BNQ algorithm.

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). Figure 2.5 gives the site cost estimation function. The actual cost calculated in Figure 2.5 is the sum of two terms. The first term is the number of queries of the same type at the site, and the second term is the number of queries of the other type multiplied by a small constant. This cost estimation function ensures that, in the case that more than one site has the same smallest number of queries of the same type , the query is routed to the site (or one of the sites) with the smallest total number of queries.

To classify a query as being I/O- or CPU-bound, its I/O demand per disk, defined as its I/O 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. If the I/O demand is greater, it is an I/O-bound query; otherwise it is CPU-bound.

```
function SiteCost(q: query; s: site): integer;
const
  Epsilon = 0.001;
begin
   if (disk_time / num_disks) > Page_CPU_Time(q) then begin
      SiteCost := Num_IO_Queries(s) + Epsilon*Num_CPU_Queries(s);
   end else begin
      SiteCost := Num_CPU_Queries(s) + Epsilon*Num_IO_Queries(s);
   end;
end;
```

Figure 2.5: Cost estimation function for BNQRD algorithm.

2.2.3. Least Estimated Response Time

The second information-based heuristic, LERT ("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, routing 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.

The response time computation itself, given as the site cost estimation function in Figure 2.6, is based on several simplifying approximations: First, it is

```
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 2.6: Cost estimation function for LERT algorithm.

assumed that competition for a given resource type at a site is only with those queries at the site that rely most heavily on this type of resource. (That is, a query will compete mainly with I/O-bound queries for I/O service at a site, and it will compete mainly 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 processor sharing (PS) service disciplines. This is probably 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 one at a time by a disk, perhaps using a SCAN ("elevator" algorithm), shortest seek time first (SSTF), or first-come first-served (FCFS) discipline [PeS83], 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.

2.2.4. Discussion

The three algorithms described in this section represent three different approaches to dynamic query allocation. The BNQ approach is based only on the load distribution (the number of queries at each site), whereas the BNQRD and LERT approaches are based on more load distribution information (the numbers of IO-bound and CPU-bound queries at each site) and information about the arriving query. Thus, only the latter two approaches use information

about the resource demands of queries. It is expected that BNQRD and LERT will 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 and returning its results to the site of origin, into account. The inclusion of these costs should make the LERT algorithm less likely to call for unprofitable query transfers than the other algorithms. LERT also uses somewhat more information about the arriving query (i.e., its estimated number of page accesses and result size). Thus, one could expect that LERT should outperform BNQRD, and BNQRD should outperform BNQ. However, a performance study is needed to investigate the degree to which this is true.

2.3. MODELING A DISTRIBUTED DATABASE SYSTEM WITH FULL REPLICATION

In order to evaluate the performance of different dynamic query allocation algorithms, a queuing network model of a database system with full replication is needed. Figure 2.7 depicts the model used here. The system consists of a collection of database processing sites, or DB sites, each of which stores a complete copy of the database. The DB sites are connected together by a local area communications subnetwork. Each site has a collection of terminals from which queries originate and to which the results are returned, so the model is a closed queuing network model. The system is assumed to be completely homogeneous, i.e., all sites are configured identically and have the same workload characteristics.

Figure 2.8 shows the detailed model of a DB site. Each site has a set of terminals, several disks, a CPU, and a number of queues. The diamonds in the

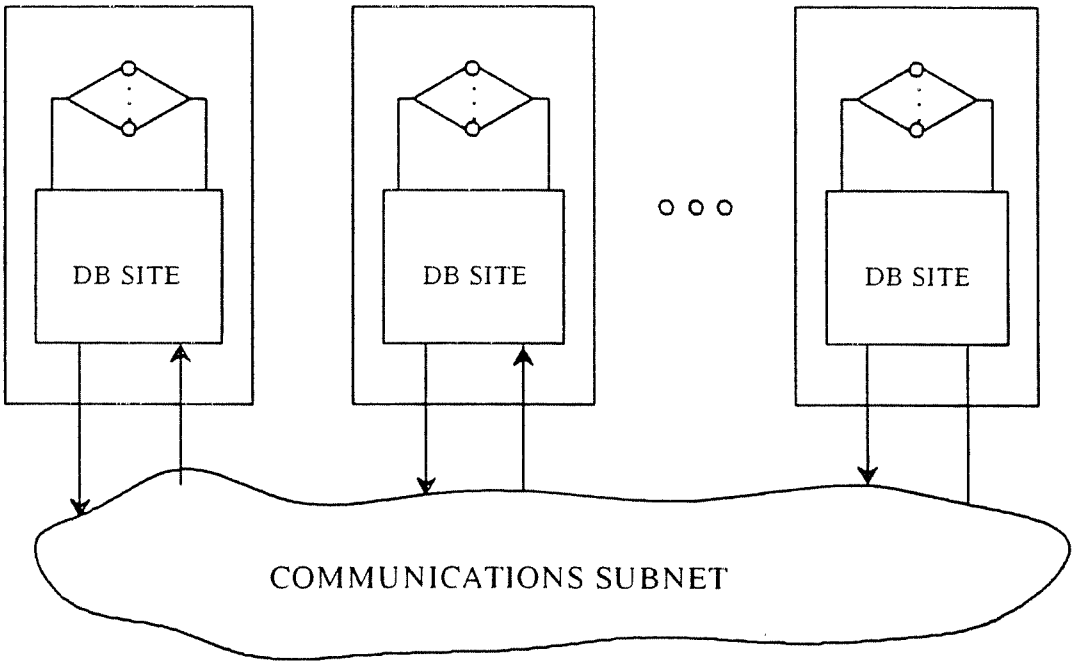

Figure 2.7: Distributed database system model.

figure depict decision points in the model. The decisions that are related to the dynamic query allocation algorithm 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 is routed to this center either directly or via the communications 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

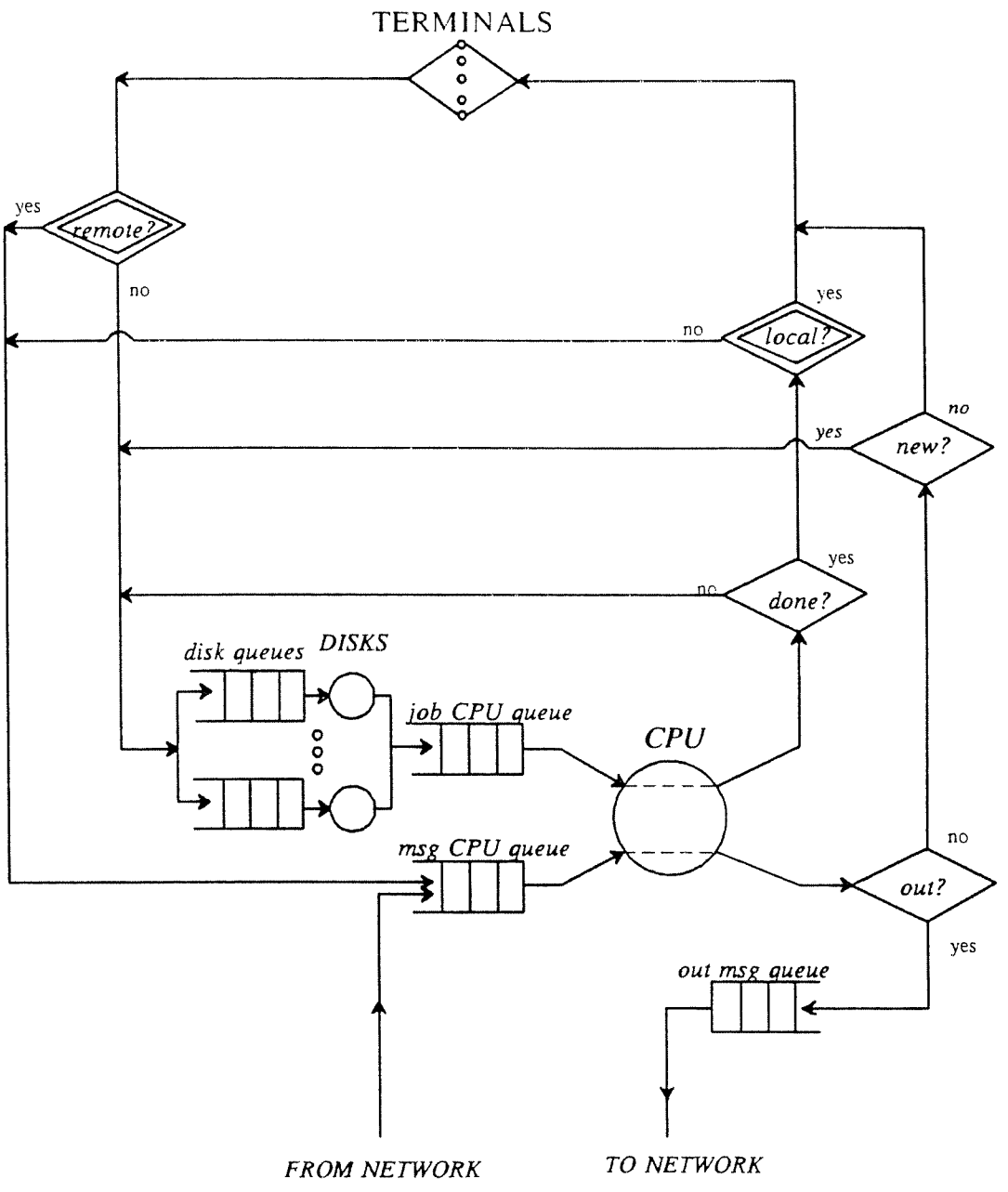

Figure 2.8: DB site model.

database.[†] 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. In order to initiate a query at a remote site or to return the query result back to its home site, messages are passed around the communications subnet. In addition to the use of the communication lines, the messages also consume CPU time (for preparing the message, executing the protocol, etc). At each site, therefore, all incoming and outgoing messages enter a *msg_cpu_queue* to receive a certain amount of CPU service. The outgoing messages then enter the *out_msg_queue* to be served by the communications subnet. An incoming message is directed either to the disk service center for execution (if it is a remote query to be executed at this site) or to the terminal (if it is a query result sent back by the remote execution site).

The scheduling discipline for the disks is FCFS (first come, first served). The CPU server serves jobs from two queues — one for query processing and the other for message processing. The scheduling discipline is PS (process sharing) for query processing and FCFS for message processing. Between these queues, message processing is given higher priority over query processing. That is, if a message arrives at the *msg_cpu_queue* while the CPU is processing queries, the query processing service will be preempted, and the message will receive CPU service immediately. The preempted query is resumed after all queued messages have been served.

Table 2.1 summarizes the parameters associated with the DB sites in the model. The storage hardware at each DB site is described by two parameters.

[†] It is assumed that the results of each query are accumulated in main memory as the query executes.

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 2.1: DB site 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 (for each class C); and *think_time*, the mean think time for the terminals.

Queries are classified into two classes in this study. Each query class Q_j is characterized by parameters $page\_cpu\_time_j$, num_reads_j, and $result\_fraction_j$. Table 2.2 summarizes these class parameters. The *page_cpu_time* parameter is the mean CPU time required to process 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

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

Table 2.2: Class parameters.

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).

The 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 along the communication line 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.

The main component of the message delay in real systems based on local area networking techniques is not the transmission delay over the communications line, but the time needed to establish a connection between two sites and to prepare the message packets [Cher83]. This preparation includes copying the data from memory to the network interface and vice versa. Even if the network protocol is supported by a memory access mechanism, message sending and receiving still causes a certain amount of CPU overhead. Therefore, the total cost for sending a message between two sites is simulated as:

$$total\_msg\_time = msg\_setup + (msg\_cpu\_time + trans\_rate) \cdot data\_length$$

The communications-related parameters are listed in Table 2.3. The parameter *msg_setup* is a fixed time delay for each message. The parameter *trans_rate* is the time needed to transmit one byte of data along the communications line. These first two parameters represent network time. *Msg_cpu_rate* is

Communications-related parameters	
msg_setup	fixed amount of time needed to establish a connection
msg_cpu_rate	CPU time needed for transferring one byte of data
trans_rate	time needed for transferring one byte over the network
query_descrip_size	size of one query message in bytes

Table 2.3: Communications-related parameters.

the CPU time needed for transferring one byte of data. The parameter *query_descrip_size* is the number of bytes required to describe a query (i.e., the amount of data that would have to be transferred to initiate a query remotely). One assumption that should be mentioned is that the overhead associated with load status messages is neglected, as it is assumed for the purpose of this study that each site simply knows the current loads of all other sites in the system.

2.4. A SIMULATION STUDY

This model of a fully replicated database system has been implemented to support investigations of the behavior of different dynamic query allocation heuristics. The details of the implementation, the experiments conducted, and the results of the experiments are all presented in this section.

2.4.1. Simulation Details

The simulation was implemented in the DISS simulation language [Melm84] and run on an IBM 4341. DISS is a high level simulation language based on the SIMSCRIPT II.5 simulation language.

In the experiments reported here, the model parameters used are those values listed in Table 2.4. The probability that a query is an I/O-bound query is

System Parameters		
num_sites	2 - 14	
DB Site Parameters		
num_disks	2	
disk_time	25 msec	
disk_time_dev	20 %	
mpl	5 - 45 terminals	
think_time	4.0 - 18.0 sec	
$class_{io}$*_prob*	0.2 - 0.8	
Communications Costs		
msg_setup	150 μsec	
msg_cpu_rate	2.0 μsec/byte	
trans_rate	10 MB/sec	
query_descrip_size	2048 bytes	
Class Parameters	I/O Bound	CPU Bound
page_cpu_time	1.25 msec	25 msec
num_reads	20 pages	20 pages

Table 2.4: Parameter settings for the simulations.

$class_{io}$*_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* ± *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 parameter *result_fraction* is the mean of an exponential result size distribution. For the parameters which are shown in Table 2.3 as varying over some range, their values when not being varied are as follows: *num_sites* = 6, *mpl* = 20, $class_{io}$*_prob* = 0.5, and *think_time* = 10.0 seconds.

2.4.2. Performance Metrics

Two performance metrics are of primary interest in this study: the mean waiting time for queries and the fairness of the algorithms. The mean waiting time (or queuing time) for the queries, $\overline{W}(x)$, is the difference between the mean response time of the queries, R, and their mean execution time, x:

$$\overline{W}(x) = R - x$$

Although response time itself is a widely used performance metric, the waiting time is preferred over response time here. This is because the response time of the queries has a fixed component, their execution time, which no query allocation algorithm can affect. What the query allocation algorithms can affect is the waiting time component of response time. The waiting time should thus better indicate the different behavior of the different query allocation algorithms.

In order to quantify the performance improvements provided by the various dynamic query allocation algorithms, the notion of the mean waiting time improvement factor, $WIF(L_1, L_2)$, is introduced. $WIF(L_1, L_2)$ is the mean waiting time improvement factor of allocation algorithm L_1 with respect to L_2, and is defined as:

$$WIF(L_1, L_2) = \frac{\overline{W}_{L_2}(x) - \overline{W}_{L_1}(x)}{\overline{W}_{L_2}(x)}$$

In this study, the dynamic allocation algorithms are compared with the static allocation algorithm LOCAL, where every query is processed locally at its home site. For example, the waiting time improvement factor of the algorithm BNQ can be represented by

$$WIF(BNQ, LOCAL) = \frac{\bar{W}_{LOCAL}(x) - \bar{W}_{BNQ}(x)}{\bar{W}_{LOCAL}(x)}$$

Another metric of interest is related to the "fairness" of the allocation algorithms. The expected waiting time of a query with a given execution time (i. e., service demand) can serve as a measure of fairness for an allocation policy as follows: Two users with service demands x_1 and x_2 are likely to agree that the allocation algorithm is fair if the ratio of their respective expected waiting times is equal to x_1/x_2. The mean normalized waiting time, $\hat{W}(x)$, is defined for this purpose; it is defined as the ratio of $\bar{W}(x)$ to x. The fairness of a system with respect to two query classes C_i and C_j when allocation algorithm L is used is defined as the difference between the normalized waiting times of these two classes. That is,

$$F_L(C_i, C_j) = \hat{W}_i - \hat{W}_j$$

The closer the F_L value is to zero, the more equitable the system is being in serving the two classes of queries. The sign of the F_L value indicates which query class is being favored by the system.

2.4.3. Simulation Experiments and Results

Several experiments have been performed to investigate how the three dynamic query allocation algorithms of Section 2.1 affect performance in terms of the metrics defined above. These experiments investigate the waiting time improvement factor and the fairness of the different algorithms, and they also examine the sensitivity of the LERT algorithm to estimation errors in the service demands.

2.4.3.1. Mean Waiting Time Improvement

Examining the effects of the different algorithms on the mean waiting time $\overline{W}$ is one of the main objectives of this study. The experiments described here investigate these effects under the following conditions:

(1) Different site load situations.

(2) Different numbers of DB sites in the system.

(3) Different communications costs.

First, the behavior of the different algorithms under different system load situations is studied. One way to obtain different system loads is to vary the *think_time* of queries. Figures 2.9 and 2.10 present the results of such an experiment, giving *WIF* and the CPU and disk utilizations as a function of think time. It is evident from Figure 2.9 that each dynamic query allocation algorithm leads to a significant decrease in $\overline{W}$ with respect to local processing, clearly displaying the beneficial effects of dynamic query allocation. Comparing Figure 2.9 with Figure 2.10, it can be seen that greater improvements are achieved when the system utilization is lower. The reason for this is that when the utilization is low, there is a better chance that a newly arrived query can be allocated to a lightly loaded or perhaps even idle DB site and therefore encounter minimal queuing delays. As anticipated, the BNQRD and LERT algorithms outperform BNQ by using additional information about query resource demands and the distribution of the query classes at each site. Finally, the performance of LERT and BNQRD is seen to be almost the same here. That is, LERT's more detailed information about the newly arrived query and its consideration of the communications cost in detail does not appear to help for the situation tested here.

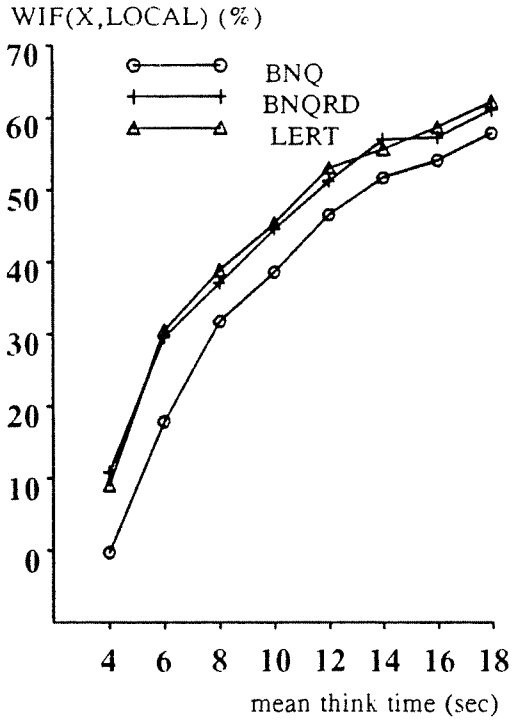

Figure 2.9: Waiting time improvement factors.

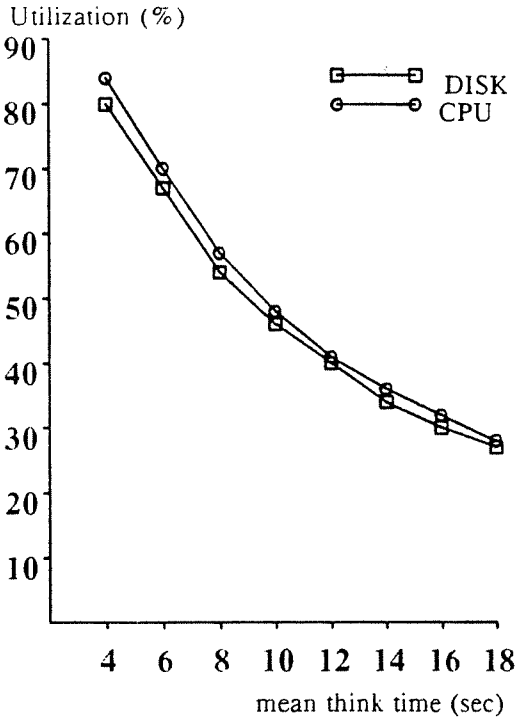

Figure 2.10: Utilizations.

Various system loads can also be obtained by varying the number of terminals (the *mpl*) at each DB site. Figures 2.11 and 2.12 show the effect of *mpl* on *WIF* and system utilization, respectively. The trends observed in this experiment are similar to those observed previously. From a multiprogramming level viewpoint, the effect of dynamic query allocation is that the number of terminals at each of the DB sites can be increased over the local processing case 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 illus-

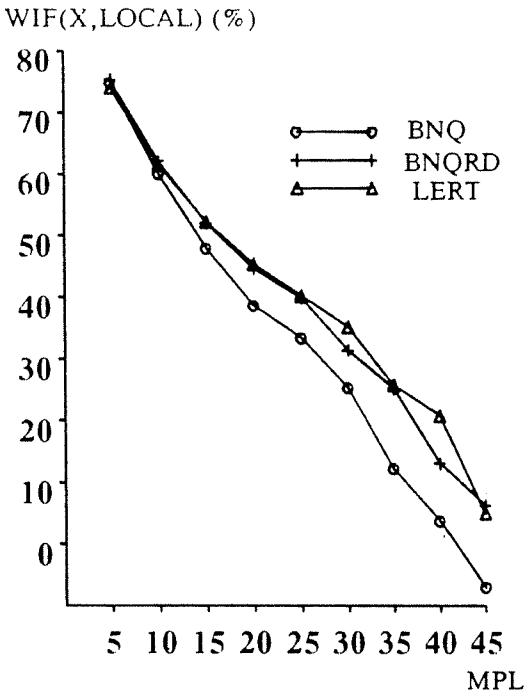

Figure 2.11: Waiting time improvement factors.

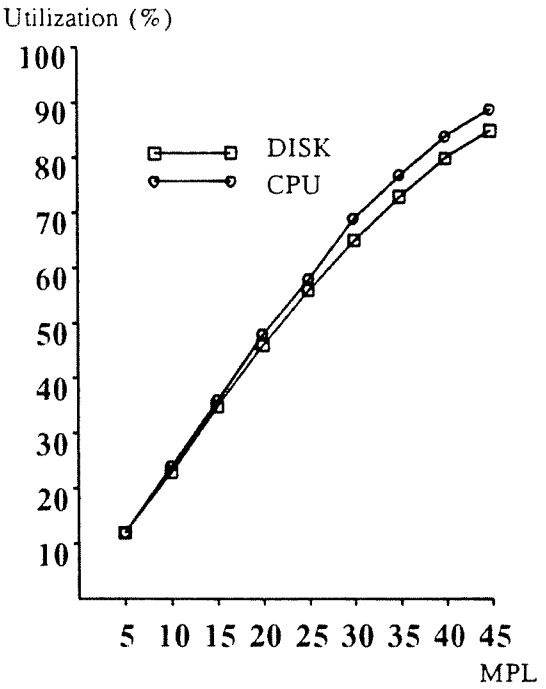

Figure 2.12: Utilization.

trated in Table 2.5, which shows the maximum number of terminals for which various expected normalized waiting times can be provided using two different allocation algorithms (LOCAL and LERT). Using LERT, the number of terminals at each site can be increased while providing the same normalized waiting time as the system with fewer terminals when queries are always processed locally.

Figures 2.13 and 2.14 summarize the results of an experiment which examines the effect of the number of DB sites in the system (*num_sites*) on

Normalized Waiting Time	Maximum *mpl*	
	LOCAL	LERT
≤ 0.20	10	16
≤ 0.40	15	22
≤ 0.90	25	31
≤ 1.25	30	35
≤ 1.80	35	38
≤ 2.50	40	42

Table 2.5: Maximum number of terminals versus $\hat{W}$.

WIF. 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 can be allocated to an idle or a lightly loaded site. This is visible in Figure 2.13 — the waiting time reduction due to dynamic allocation increases as *num_sites* is varied from 2 to 12. On the other hand, however, it increases competition for the communication channel and thus increases the waiting time for messages. To illustrate this phenomenon, Figure 2.14 shows the subnet utilization for each of the *num_sites* settings. Due to the combination of these two effects, the waiting time improvement factor is approximately constant for *num_sites* values greater than 10. Even the improvement for LERT decreases when the number of DB sites increases. Although LERT takes message costs into account, it does not consider the waiting time for messages caused by the heavily loaded communication line. If the communication cost were higher, the value of WIF would actually begin to decrease after a while as *num_sites* is increased [Care85]. This result 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, and an important

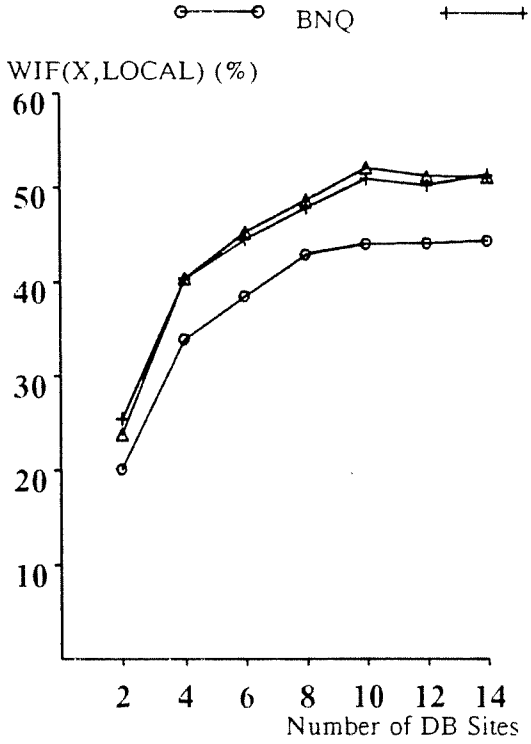

Figure 2.13: Waiting time improvement.

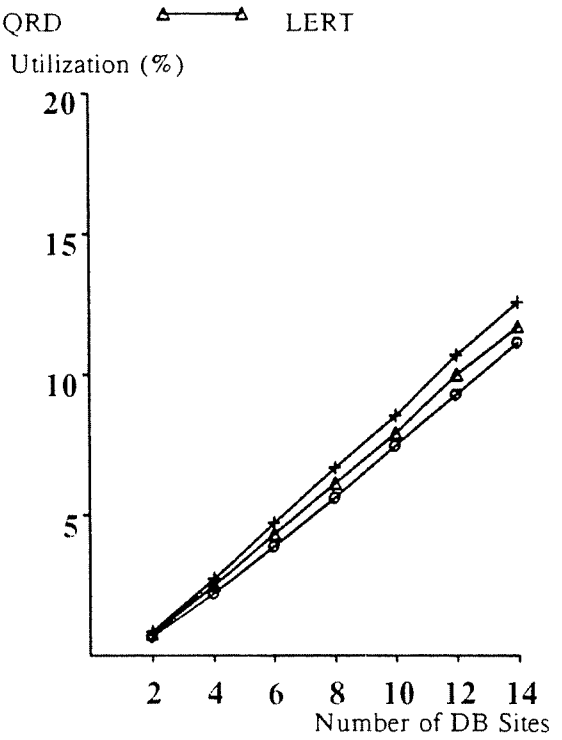

Figure 2.14: Subnet utilization.

parameter affecting this value is the data transfer rate of the communications network. This observation coincides with the idea of selecting a suitable load balancing region, as discussed in Chapter 1.

Another experiment conducted investigates the improvement in the mean waiting time for queries under different communications costs. In order to vary the communications cost, the result fraction of the query was varied. Figure 2.15 shows the results of this experiment. With a small result fraction, say

0.01, only one packet is sent and returned for a remote query. When the result fraction increases, the size of the result becomes larger. With a result fraction of 0.3, and a *num_reads* setting of 20, the mean result size will be 6 pages. The total time needed to return these results, excluding any waiting time for the communications line, is about 118 milliseconds — or about 22% of the execution time for an I/O bound query.

The BNQ and BNQRD algorithms do not take this cost into account when they make their transfer decisions, leading to poor performance for these

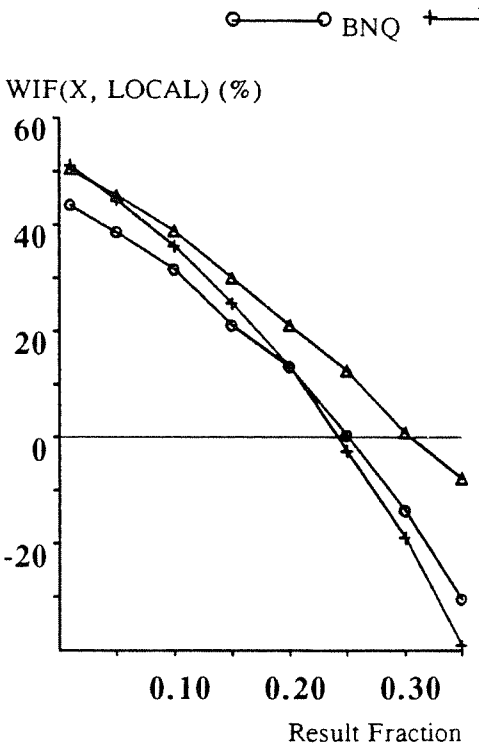

Figure 2.15: Waiting time improvement factors.

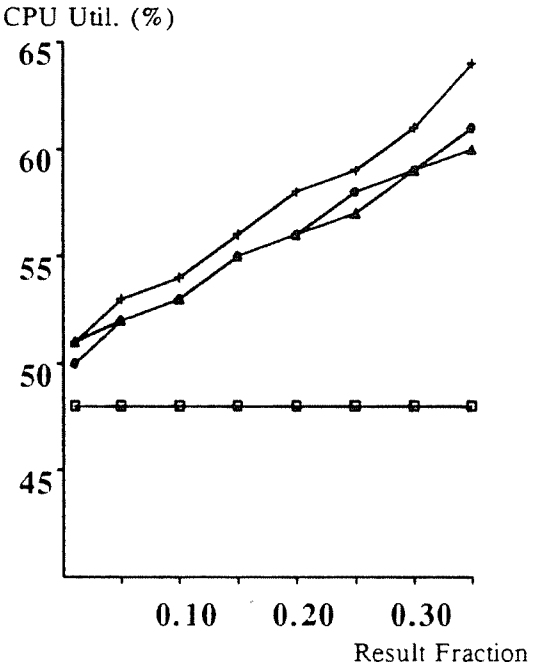

Figure 2.16: CPU utilization.

algorithms (even worse than LOCAL) when the result fraction of the query increases. The LERT algorithm, which considers this effect, thus performs better than BNQRD when the *result_fraction* increases. Still, two factors are neglected by the LERT algorithm. The first factor is the CPU time needed to process the incoming and outgoing messages. When the result fraction becomes larger, the CPU time consumed by the messages increases dramatically. This can be seen in Figure 2.16, which shows the CPU utilization as a function of the result fraction. As the result fraction goes from 0.01 to 0.35, the CPU utilization increases about 15 percent; furthermore, message processing has high priority, so the query scheduling discipline for the CPU queue will not be "pure" processor sharing (which is one of the assumptions made by LERT). The second factor which is not considered by LERT is the waiting time for messages in the network. When the result fraction increases, the load on the communications subnet also increases, as shown in Figure 2.17. These two factors make LERT's estimation of response time less accurate and lead to incorrect allocation decisions. This explains why the LERT algorithm leads to somewhat poorer performance than LOCAL when the result fraction is high.

2.4.3.2. Dynamic Query Allocation and Fairness

To investigate the effects of dynamic allocation algorithms on the fairness of the system to the two job classes, another experiment was performed in which the parameter $class_{io}\_prob$ was varied from 0.2 to 0.8 (which makes the system without dynamic query allocation favor one or the other of the query classes). The results are shown in Figures 2.18 and 2.19. Along with the significant waiting time improvement, shown in Figure 2.18, an improvement in the fairness measure F_L, $\hat{W}_{io\_bound} - \hat{W}_{cpu\_bound}$, is evident in Figure 2.19 in most

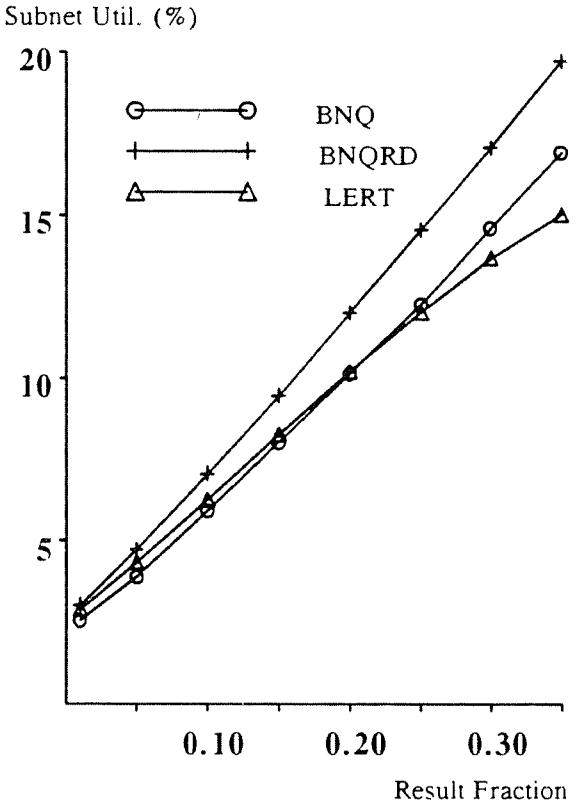

Figure 2.17: Subnet utilization.

cases. When $class_{io}\_prob$ is less than 0.5, the system favors I/O bound queries (i.e., $\hat{W}_{io\_bound}$ is less than $\hat{W}_{cpu\_bound}$). When $class_{io}\_prob$ is greater than 0.5, the system begins to favor CPU bound jobs. This can be seen from the LOCAL curve in the figure. 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 the normalized waiting times between the two classes.

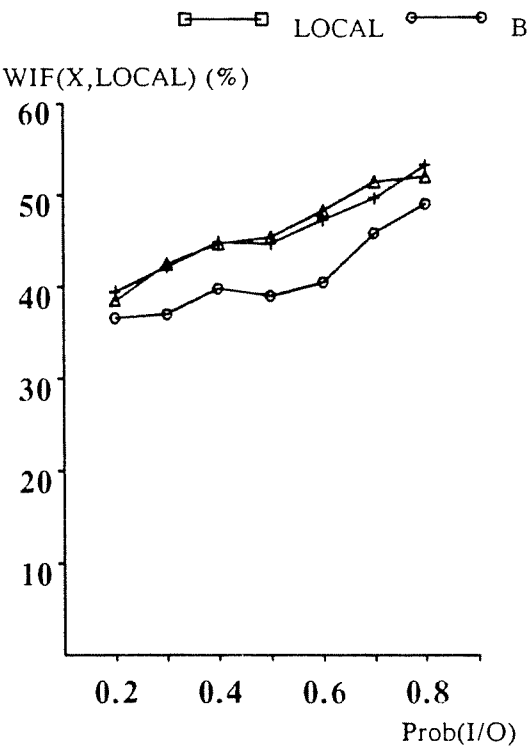

Figure 2.18: Waiting time improvement factor.

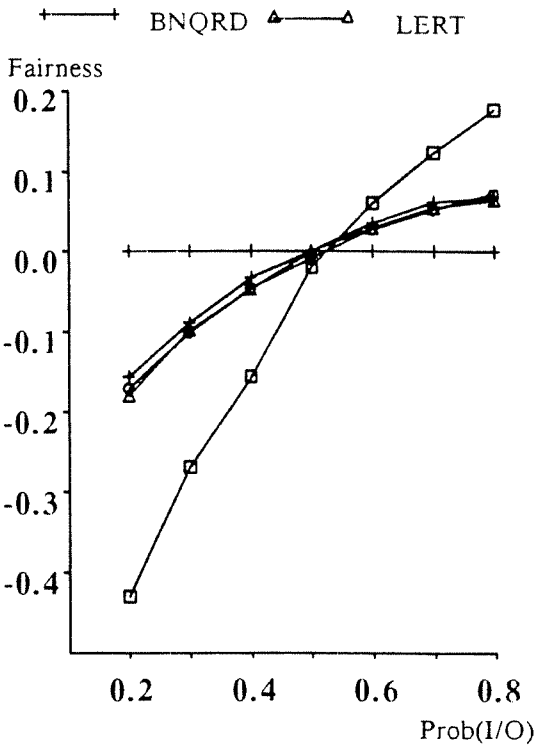

Figure 2.19: Fairness.

2.4.3.3. Sensitivity to Query Information

The dynamic allocation algorithms BNQRD and LERT use information provided by the query optimizer about the processing costs of queries. However, the query optimizer can only provide an *estimate* of the processing cost of a query. That is, the information given to the load balancing algorithm is not the actual service demand for a query. Since LERT depends more upon these service demands to make allocation decisions than BNQRD does, the inaccuracy of

the estimates is expected to have a greater influence on the performance of the LERT algorithm. An experiment was run to investigate the sensitivity of the LERT algorithm to such inaccuracies. In this experiment, instead of using perfect information about the resource demands of a query (*io_cost* and *cpu_cost*), their estimates (*est_io_cost* and *est_cpu_cost*) were used to estimate the response times in the LERT algorithm. Both *est_io_cost* and *est_cpu_cost* were assumed to be normally distributed random variables with *io_cost* and *cpu_cost* as their respective means. The mean waiting time for queries as a function of the deviation of these normal distributions is shown in Figure 2.20 for the LOCAL, BNQ, and LERT algorithms.

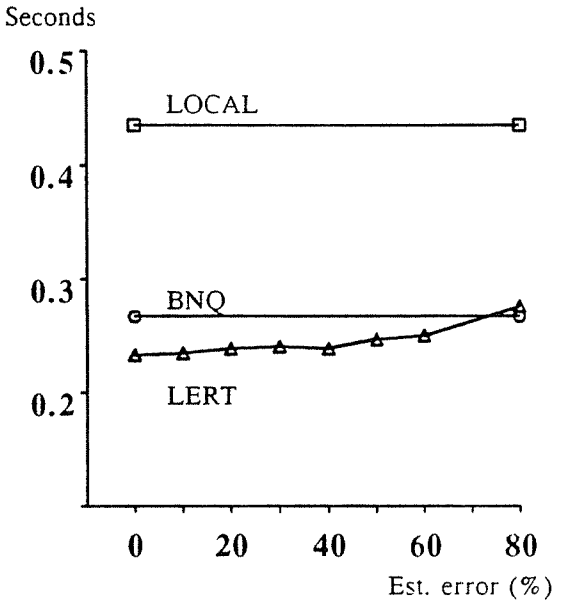

Figure 2.20: Waiting time.

The trends shown in the figure are encouraging: While the mean waiting time increases as the estimation error increases, the LERT algorithm performs better than the BNQ algorithm when the estimation error is less than 80% of the mean execution time. That is, even with quite inaccurate information about queries' resource demands, the information-based algorithm (LERT) still outperforms the non-information based algorithm (BNQ). One argument explaining the importance of accurate query processing demand estimation is that the information needed for dynamic allocation is probably not the *absolute* response time, but rather the *relative* response time for the different sites. That is, if the estimated response time for a query at site S_i is greater than that of another site S_j, and if this implies that the actual response time at S_i will be greater than that at S_j, then absolute estimation accuracy is not at all that significant.

2.5. SUMMARY

In this chapter, the problem of allocating queries to sites in a fully replicated distributed database system has been studied. Dynamic allocation algorithms which use different amounts of query and site load information were described. A closed, two-class queuing network model was established for a fully replicated distributed database system. A set of simulation experiments were conducted using this model, and the results indicate that significant decreases in the mean waiting time for queries can be obtained when queries are allocated to sites based on load information. Another important conclusion of this study is that using information about query resource demands leads to significantly better performance than the simple 'balance the number of queries' approach. It was found that the overall fairness of the system can be improved through dynamic query allocation. Finally, it was also found that the query

resource demand estimates do not have to be exact to yield improvements in performance.

The environment in which these dynamic allocation algorithms have been studied is that of a distributed database system with both CPU and I/O (disk) resource demands. However, database systems are not the only application in which I/O as well as CPU usage 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, and this information could be used by a distributed operating system to make more informed load balancing decisions, as in the database environment.

Finally, it should be pointed out that nothing has been mentioned about *how* site state information is to be exchanged among the sites — this omission is intentional. This study has 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 the performance improvements offered by the heuristics are not lost. The design and analysis of such a policy is not addressed in this thesis, being left instead for future work.

CHAPTER 3

DISTRIBUTED JOIN ALGORITHMS: AN EMPIRICAL STUDY

Compared to the number of distributed query processing *algorithm* papers that have appeared in the literature, relatively few papers have addressed the *performance* of distributed query processing algorithms. In order to get a deeper understanding of distributed query processing issues in local networks, an empirical performance study of different distributed join methods was conducted using Crystal [DeWi84b], an experimental locally distributed computer system. This chapter describes these join methods, the testbed built for the experiments and the implementation details. The results obtained are then presented, and their implications and relationship to other work are discussed at the end of the chapter.

3.1. DISTRIBUTED JOIN METHODS

Given an equi-join query $R_a \bowtie R_b$ [†] in a distributed database system, where R_a and R_b reside at (different) sites S_a and S_b, respectively, there are a number of distributed join methods available for processing it. These methods can be categorized along three dimensions.

(1) General approach: Which primitive operator is used — the "traditional" join operator or the semijoin operator?

(2) Execution paradigm: How does the pair of sites involved cooperate during join processing — in a sequential fashion or in a pipelined fashion?

[†] $R_a \bowtie R_b$ is used as an abbreviation for $R_a[A=B]R_b$ here.

(3) Local processing. What access paths and local join methods are used for local data access and processing?

3.1.1. Join Versus Semijoin

As described in Chapter 1, a distributed join can be processed either using the traditional join operator or the semijoin operator. If the traditional join operator is used, the two relations must be brought together at one site. This can be done by either transferring one relation to the site where the other relation is stored or by transferring both relations to a third site. In either case, the whole relation is transferred. Using the semijoin operator, the join column values of one relation, say $R_a.A$, are transferred to the site where the other relation R_b resides to perform the semijoin $R_a \rtimes R_b$[†]. The join $R_a \bowtie R_b$ is completed by sending the semijoin results to the site of R_a and joining them with R_a. If inter-site data transfers are expensive, the join field width is small compared to the width of an entire tuple, and the semijoin selectivity between the two relations is small (i.e. there are not many matching tuples), the use of semijoins can result in a significant savings in communications cost. In local area networks, however, the data transfer rate between sites is much higher. It is questionable whether or not semijoins will be beneficial in such an environment, as using them requires multiple scans of the source relation R_a and therefore more disk accesses.

3.1.2. Sequential Versus Pipelined Processing

The concept of pipelined processing is widely used. In query processing, the partial result of one operation can be used as the input of the next operation

[†] $R_a \rtimes R_b$ is used as abbreviation for $R_a <A = B] R_b$ here.

to speed up the processing [Smit75][Yao79]. The same concept can be applied to the processing of a single join operation, $R_a \bowtie R_b$, where R_a and R_b are at two different sites. That is, the site that requests remote data will begin its processing as soon as the first tuple or packet of data has arrived. This is in contrast to sequential processing, where the site receiving data will not begin its processing until all of the required data has arrived. One advantage of the pipelined approach is its parallelism — the two sites work in parallel, so the elapsed time for the query will be reduced in proportion to the amount of overlapped processing. Second, and perhaps more important, is the fact that the receiving site doesn't need to actually store the incoming data in a temporary relation, thus saving the time and disk accesses required to store and then re-retrieve the data received from the remote site. Pipelined processing can be applied to both join-based and semijoin-based join processing. Its main drawback is that flow control is needed to synchronize the processing at two sites, so its implementation is somewhat more complicated than that of sequential processing.

3.1.3. Access Paths and Local Join Methods

Since any distributed join involves local processing, the local join algorithm and associated access methods are still important factors in distributed join processing. There are a number of options available along this dimension. For centralized join methods, Blasgen and Eswaran found that, except for very small relations, the nested loops join or sort-merge join methods were always optimal or near optimal [Blas77] As for access methods, the two major options are sequential scan or an index scan using various index structures. Recently, it is reported that even better performance can often be obtained using hash-based join methods [DeWi84a]. Only the nested loops and sort-merge methods are

examined here, however.

In this study, the availability of an unclustered B^+ tree index on the join column of the inner relation is assumed in all experiments with the nested loops join method.[†] An unclustered index is assumed because it is not always reasonable to assume that the join column will have a clustered index available. If the inner relation is shipped to the outer relation's site, an unclustered B^+ tree index is constructed for the inner relation at the outer site. An unclustered is used here because it is faster to build an unclustered index than a clustered one. Furthermore, since the join column values of the outer relation are not ordered, clustered and unclustered indices will produce the same performance.

3.2. JOIN ALGORITHM DETAILS

Eight algorithms were implemented for this study. In terms of the three dimensions described above, the algorithms are the eight possible combinations of the semijoin-based or join-based approach, the pipelined or sequential execution st es, and the nested loops index join or sort-merge local join methods. These methods are abbreviated as SJSM, SJNL, PJSM, PJNL, SSSM, SSNL, PSSM, and PSNL respectively. For the first letter, "S" stands for sequential and "P" stands for pipelined. The second letter, "J" or "S", is used to represent "traditional"join versus semijoin. The last two letters indicate either sort-merge ("SM") or nested loops ("NL") join. In all tests, site S_a initiates the join query, and S_a is both the join site and the result site. In the remainder of this section, each distributed join method is described in turn.

[†] The cost of a nested loops join without an index for relations of reasonable size is usually prohibitive [Bitt83], so this possibility is not considered.

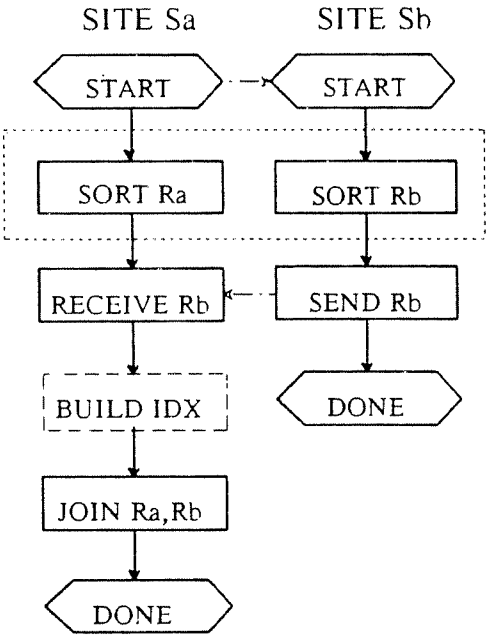

Figure 3.1:
Join methods SJSM and SJNL.

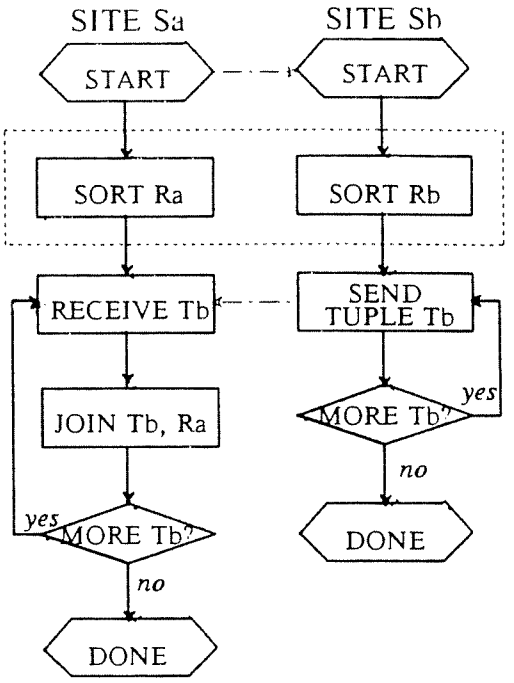

Figure 3.2 :
Join methods PJSM and PJNL.

3.2.1. SJSM and SJNL

The sequential join methods SJSM and SJNL are the simplest of the distributed join methods. Figure 3.1 illustrates the details of the two methods. The remote relation R_b is shipped to join site S_a as a whole and stored there as a temporary relation. It is then joined with R_a at site S_a using either the sort-merge method (SJSM) or the nested loops method (SJNL). For SJSM, the two relations are each sorted at their local sites to increase parallelism (as shown in the dotted box); no sorting is performed in the case of SJNL. For SJNL, a B^+

tree index is built on the join column of the relation received at site S_a before the local join is performed (indicated by the dashed box in the figure).[†]

3.2.2. PJSM and PJNL

Like SJSM and SJNL, the PJSM and PJNL algorithms transfer the whole relation R_b from site S_b to site S_a. However, the joins are processed in a pipelined fashion, so relation R_b is not stored as a temporary relation at site S_a. Instead, tuples of R_b are joined with R_a tuples on the fly as they arrive, as shown in Figure 3.2 (with the sorting steps only being present in the sort merge case). Using the PJSM method, both relations are sorted first. Then, when a group (packet) of R_b tuples arrives at site S_a, a scan cursor on R_a is incremented to find matching tuples. Matches are merged with the tuples from the R_b group and written to the result relation. In the PJNL case, since no temporary relation for R_b is stored at site S_a, the local relation R_a always serves as the inner relation and the remote relation R_b serves as the outer relation. Both PJSM and PJNL can be viewed as distributed executions of the corresponding centralized join algorithms.

3.2.3. SSSM and SSNL

Figure 3.3 illustrates the details of two implementations of the semijoin-based methods, SSSM and SSNL. One important detail of the implementation is that the join column $R_a.A$, which is sent from site S_a to site S_b, is not stored on disk at site S_b — the incoming values are processed on the fly as they arrive. Similarly, the semijoin result $R_b' = R_a.A \rhd\lhd R_b$, which is transferred back to

[†]One possible local optimization for sort-merge methods is to complete the last phase of sorting on the fly: the sorted tuples are not stored and sent to the network directly.

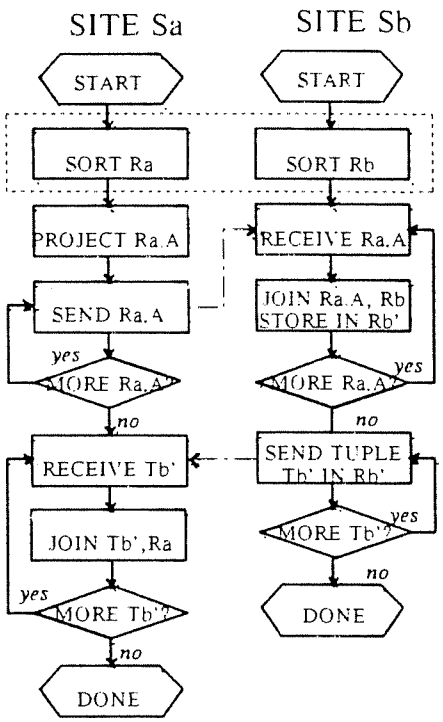

Figure 3.3:
Join methods SSSM and SSNL.

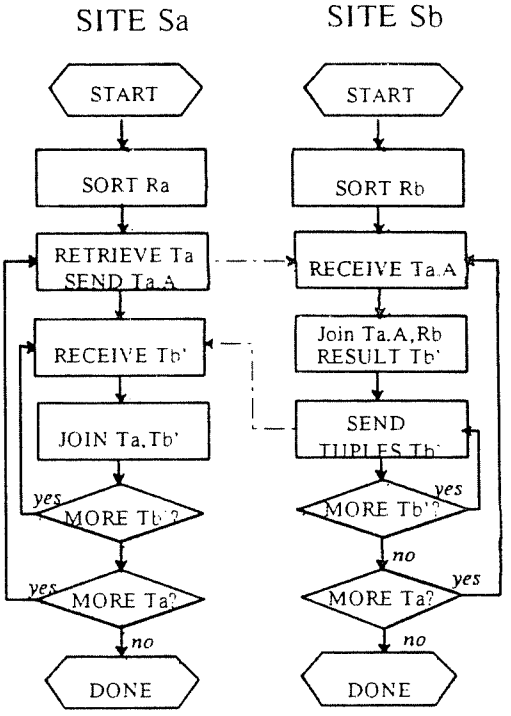

Figure 3.4:
Join methods PSSM and PSNL.

site S_a, is also processed on the fly as it arrives at S_a instead of being stored there as a temporary relation. Although there is therefore some limited pipelining involved in the SSSM and SSNL execution strategies, they are still categorized as being sequential as compared to the much more pipelined PSSM and PSNL algorithms to be described next.

3.2.4. PSSM and PSNL

The pipelined semijoin methods, referred to as "fetch inner tuples as needed" in System R^* [Seli80], were the most complicated of the eight join methods to implement. As shown in Figure 3.4, relation R_a is scanned in a tuple-by-tuple manner (conceptually), and join column values $R_a.A$ are sent to site S_b. Upon receiving an $R_a.A$ value, site S_b selects the matching tuples from R_b and sends them to S_a; a null message is sent if there are no matching tuples. These tuples are then merged with the corresponding tuple of R_a, which is waiting for them (still in main memory). However, the implementation used here is a little different from System R^* method. R_a tuples are actually processed in one-page batches, so one buffer page is allocated for holding the tuples from R_a.

3.2.5. Discussion

The eight distributed join methods described in this section represent a range of possible methods. The sequential join methods, pipelined join methods, and pipelined semijoin methods are all among the methods used by System R^* [Seli80][Lohm85], although the implementation here may differ in minor ways. Of these methods, the sequential join methods are attractive for their simplicity, while the pipelined methods allow more concurrency and avoid the cost of storing and retrieving tuples from a temporary relation. The pipelined methods, of course, require some synchronization of the two processing sites (in the form of flow control, so the receiving site can indeed avoid having to store incoming tuples). One resulting limitation of PJNL (the pipelined nested loops join method) is that R_a must be the inner relation, regardless of

how its size compares with that of R_b, as the inner relation has to be available for multiple scans. The semijoin methods are expected to reduce communications costs. The main difference between the pipelined and sequential semijoin methods is related to duplicates — since the pipelined versions simply scan R_a instead of projecting on $R_a.A$, they will send duplicate join column values if any are present in R_a; however, the sequential semijoin methods require multiple scans of R_a, increasing the local processing cost. Clearly, there are tradeoffs among all of these algorithms.

3.3. THE EXPERIMENTAL TESTBED

Figure 3.5 depicts the testbed system used for this performance study. A collection of test programs was written to implement (hard-wired) distributed join queries using the different methods described in section 3.2. These programs access a synthetic database, the Wisconsin database [Bitt83], via WISS, the Wisconsin Storage System, [Chou85]. The programs run on a pair of node machines from the Crystal multicomputer, an experimental distributed computer system [DeWi84b]. Monitor programs run on a VAX/UNIX[†] host machine to initiate test program execution and to collect performance statistics after the test programs terminate. For communications between the node machines, or between the node machines and the host, a Crystal communications package called the Simple Application Package (SAP) is used. In this section each of these components of the system is briefly described.

[†] UNIX is a Trademark of AT&T Bell Laboratories.

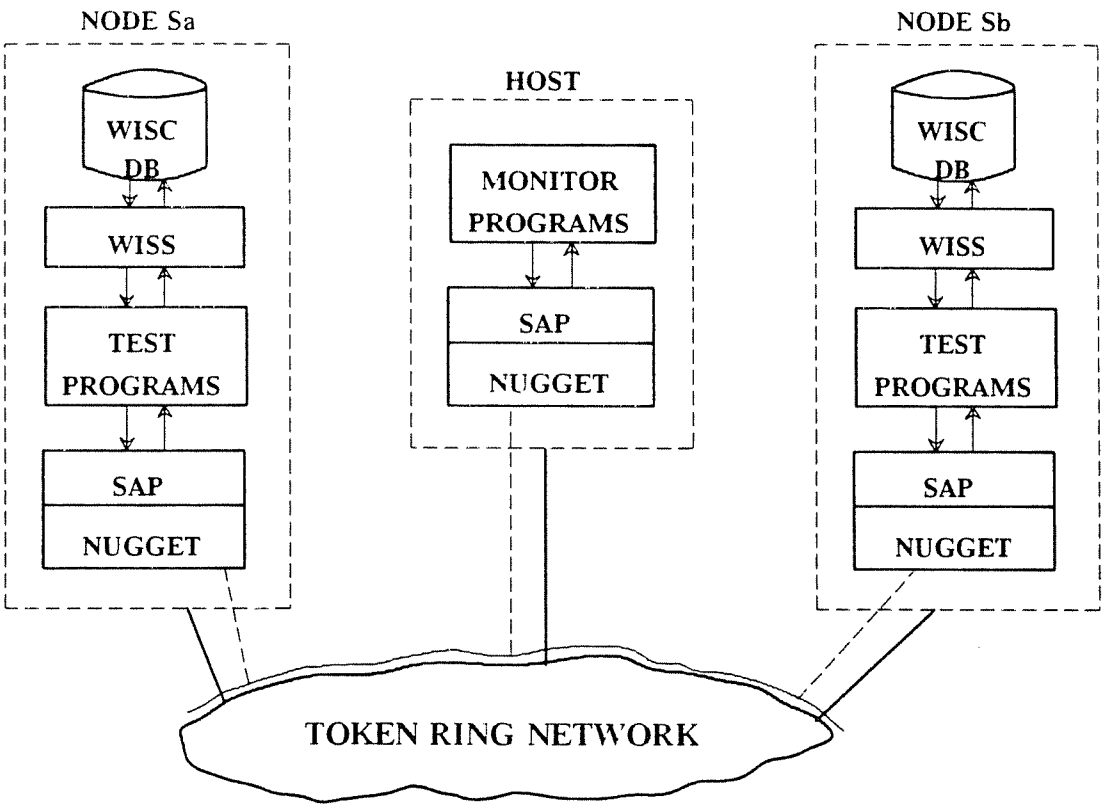

Figure 3.5: The testbed for distributed join methods.

3.3.1. The Crystal Multicomputer

At the time of these experiments, the Crystal multicomputer [DeWi84b] consisted of 20 DEC VAX 11/750's interconnected via a 10-Mb/sec Proteon token ring network. The Crystal ring network is also connected to several of the Computer Sciences Department's research VAXes, each of which can serve as a host machine. Crystal multicomputer users can claim a number of node machines as a partition. The partitions of different users in the system are

logically isolated from each other — each partition is basically a virtual distributed computer system. A partition of two node machines with 2 megabytes of memory and 160 megabyte Fujitsu disks was used to create the distributed database system testbed for this study.

There are several levels of software available on Crystal. To avoid unnecessary overhead, only two of the lowest levels, the Crystal Nugget and the Simple Applications Package (SAP), were used in the experiments. The Nugget is a simple communications kernel that resides permanently on each node machine and host machine, providing low-level message-passing primitives and ensuring that no messages are sent to nodes outside the user's partition (i.e., enforcing the logical isolation of partitions). SAP is a set of subroutines that sit on top of the Nugget, providing buffered communications using two queues, one for incoming messages and other for outgoing messages. With SAP, the message sender busy-waits until a send buffer is available, fills the buffer with data, indicates its destination, and then invokes a non-blocking send routine. To receive a message, the receiver busy-waits on a flag that indicates that the input buffer is full. The flag is set when a message is received from the Nugget and reset when it is consumed by the user's program. Thus, SAP provides a somewhat higher-level message facility than the Nugget. The message packet size is 2K bytes, including 60 bytes of control information for the experiments.

3.3.2. The Wisconsin Storage System

The Wisconsin Storage System, or WISS, is an access-method level data storage system that can run either on top of UNIX or directly on top of a "raw" disk [Chou85]. When used on top of a raw disk, WISS implements an extent-

based file system. For this study, WISS was installed on the Crystal node machines and accessed their disks directly. WISS has a layered structure that consists of four levels — physical I/O , buffer management, storage structure and access methods. The physical I/O management level is responsible for reading pages from and writing pages to the disk. The buffer management level maintains pools of buffer pages. One such pool is associated with each WISS user. When the buffer pool is full, the buffer manager makes a replacement decision using an LRU replacement strategy and some hints from the system about which pages are important. These hints give preference to pages such as B^+ tree root pages and system directory pages. The storage structure level of WISS is responsible for implementing a record-level storage abstraction. Sequential files, long data items, and B^+ tree indexes (clustered or unclustered) are all provided as structured files by this level of WISS. Finally, the highest level of WISS, the access method level, implements the access methods of sequential scan, index scan, and long data item scan and provides control over scans (such as the capability to reset a scan cursor). This level also provides routines for creating and destroying files, indexes and long data items. The distributed join programs interface with WISS mainly at this level.

3.3.3. The Wisconsin Database

The Wisconsin Database was designed for use in systematically benchmarking relational database systems [Bitt83]. There are four basic relations in the database, "thoustup", "twothoustup", "fivethoustup", and "tenthoustup", which contain 1000, 2000, 5000, and 10,000 tuples, respectively. Tuples in all of these relations are 182 bytes long, each consisting of thirteen 2-byte integer attributes and three 52-byte string attributes. All of the integer attributes have

unique1	unique2	two	four	ten	thousand	fivethous	...
1347	6709	0	0	0	937	929	...
9354	1591	1	1	4	985	1762	...
1595	2651	0	2	3	829	1967	...
3806	4474	1	3	2	936	2923	...
8727	1930	0	0	1	820	3849	...
9282	1293	1	1	5	187	1042	...
8124	885	0	2	8	198	4737	...
3228	9584	1	3	9	734	4082	...
3654	2307	0	0	6	184	1621	...
3761	2508	1	1	7	445	3231	...

Table 3.1: A fragment of the "tenthoustup" relation.

uniformly distributed values, but the range of their distributions varies. Table 3.1 shows a portion of the "tenthoustup" relation to illustrate the purpose of the different integer attributes. The "unique1" and "unique2" attributes are both candidate keys, taking on values from 0 to 9999. The other integer attributes all take on random values chosen in a way indicated by their name — for instance, the "ten" attribute takes on values from 0 to 9 (i.e., 10 distinct values). Thus, different columns of a relation can be used in queries to obtain desired selectivities, projectivities or join selectivities. The integer attributes in the "thoustup", "twothoustup", and "fivethoustup" relations are similar.

3.4. EXPERIMENTS AND RESULTS

3.4.1. Some Considerations

The first problem that arose in designing the tests was the issue of choosing an appropriate set of test queries. In their classic study of join methods for centralized database systems, Blasgen and Eswaran used a query that selected a sub-

set of tuples from two relations, joined these together, and finally projected out a subset of the resulting fields as a general query for their analyses [Blas77]. This query could also be used for a study of distributed joins. However, since the effects of the data distribution on various join methods are the main issue of interest, the simple two-site join shown in Figure 3.6 was used for the test queries. The sizes of relations R_a and R_b, the size of the result relation R, and the value distributions of the join attributes are varied in the experiments to observe their respective effects on performance. Using this simple join query does not really limit what can be learned from the study — adding the two pre-join selections and a post-join projection would only increase the fraction of the execution time due to local processing, and this time would be the same for all the distributed join algorithms.

The choice of source relations for this study followed the methodology presented in [Bitt83]. First, it was felt that relation sizes should be large enough to be realistic. The basic relations used in the tests have 1,000 tuples and 10,000 tuples (the "thoustup" and "tenthoustup" relations of the Wisconsin database),

---

```
range of a is Ra at Sa
range of b is Rb at Sb
retrieve into R(a.all,b.all) at Sa
where (a.A = b.B)
```

Figure 3.6: General form of the test query.

---

occupying about 46 and 456 pages (page size in WISS is 4K), respectively. Second, it was felt that random attribute value distributions are desirable in order to provide an unbiased treatment of each of the join methods. This is particularly important in the sort-merge join case. Third, in order to insure that the results of the various tests were not biased by previous tests, it had to be ensured that no test query was likely to find useful pages sitting in the buffer pool from its predecessors. The technique described in [Bitt83] was used, where two copies of source relations are maintained (at each site in our case), and alternate queries use alternate copies of the source relations.

Another important decision was the choice of an appropriate set of performance metrics and a reasonable measurement approach. In all experiments, the elapsed time of a query was the main metric measured. This time is defined as the time interval beginning when site S_a initiates the query and ending when the result is completely stored at site S_a. The Crystal Nugget provides a timing procedure that is accurate to the nearest 10 milliseconds; this procedure was used for the elapsed time measurements. For each query, the number of disk accesses performed and the number of messages sent were also measured. The disk access measurements were taken using a special version of WISS that is instrumented to trace disk operations [Chou85]. For each disk access, the start and completion times of the access were recorded. An analysis of the trace records from the experiments indicates that an average disk access in this test environment takes about 25.5 milliseconds (for a 4K-byte page). To measure network traffic, the number of messages were recorded. To measure the actual message send and receive times, a separate test that sent and received a large number of single-packet "null" messages between two node machines was con-

ducted. This message sending and receiving program used the same communications interface routines used for the test queries. The results indicated that the average message transfer time is about 16.6 milliseconds for a 2K-byte packet. Finally, while the CPU time used by the test queries is also an important performance metric, it was not easily measured at the level at which the experiments ran (i.e., stand-alone on Crystal nodes).

3.4.2. The Experiments and Results

The test queries were designed to investigate the effects of a number of different factors on the performance of the alternative distributed join algorithms. The factors investigated include the sizes of the source relations and the join selectivity (i.e., the result relation size and the distributions of the join column values). The experiments and the results obtained are described in this section. First, however, the results from one of the distributed join executions are described in detail in order to illustrate the costs and benefits of the various approaches and to provide the reader with useful background knowledge for later discussions.

3.4.2.1. Query Resource Demands: A Detailed Example

The example examined in this section involves a query where both R_a and R_b are "thoustup" relations and the result relation has 100 tuples.

Figure 3.7 shows the elapsed time for processing the example query using the eight different join methods. The elapsed time for site S_a is the actual elapsed time of the query, and the elapsed time for site S_b shows the portion of time during which S_b was involved in the query. The general trend is that the pipelined join methods — PJSM, PSSM, PJNL, and PSNL — executed the join

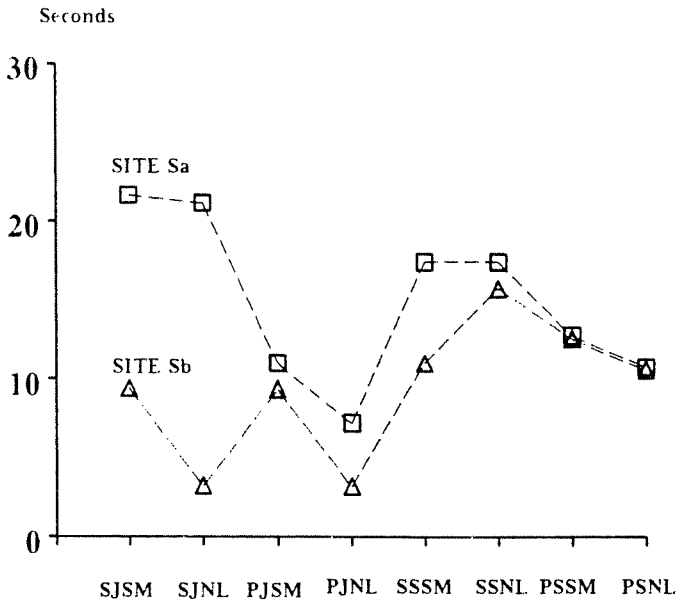

Figure 3.7: Elapsed time.

more quickly than the sequential methods did. Of the pipelined methods, the nested loops join methods outperformed the sort-merge methods for this example. (That is, PJNL did better than PJSM, and PSNL did better than PSSM). This can be explained by taking a look at the resource demands of the various join methods.

The Number of Messages:

Figure 3.8 shows the number of messages that were required to transfer data between the two sites S_a and S_b (measured at site S_a), illustrating the communications cost of each of the join methods. It can be seen that the sequential semijoin methods (SSSM and SSNL) require the least number of messages and

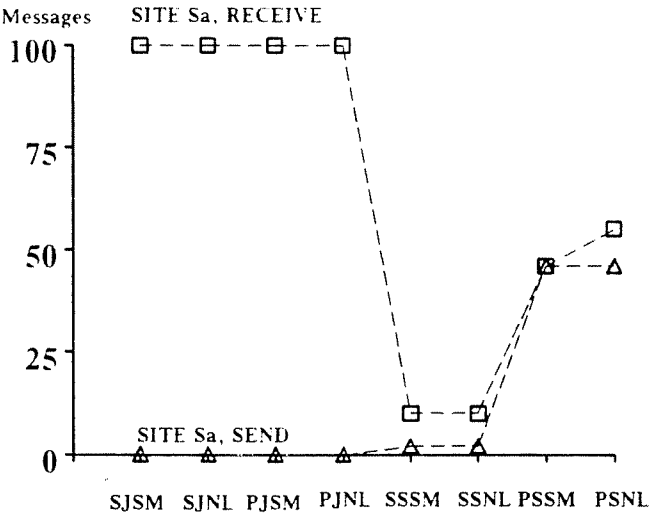

Figure 3.8: Number of messages.

that the join-based methods require the most. Table 3.2 gives estimates for the number of messages required by each join method. Before explaining the estimates in details, the *message factor*, F_M is introduced to represent the number of tuples or attribute values that can be held in one message. That is,

$$F_M(R) = \left\lfloor \frac{msg\_size}{tuple\_len(R)} \right\rfloor$$

and

$$F_M(R.A) = \left\lfloor \frac{msg\_size}{attr\_len(R.A)} \right\rfloor \tag{3.4.1}$$

For each of the join-based methods (SJNL, SJSM, PJSM and PJNL), site S_a sends no messages to site S_b, and the messages received by S_a contain the whole relation R_b. For the sequential semijoin methods (SSNL and SSSM), site

The Number of Messages (at Site Sa)		
Method	Send	Receive
SJSM SJNL PJSM PJNL	0	$\left\lceil \frac{\mid R_b \mid}{F_M(R_b)} \right\rceil$
SSSM SSNL	$\left\lceil \frac{\mid R_a.A \mid}{F_M(R_a.A)} \right\rceil$	$\left\lceil \frac{J_s(Ra,Rb)\cdot \mid R_b \mid}{F_M(R_b)} \right\rceil$
PSSM PSNL	$\left\lceil \frac{\mid R_a \mid}{F_B} \right\rceil \cdot \left\lceil \frac{F_B}{F_M(R_a.A)} \right\rceil$	$\geq \left\lceil \frac{\mid R_a \mid}{F_B} \right\rceil \cdot \left\lceil \frac{F_B}{F_M(R_a.A)} \right\rceil$

Table 3.2: Estimation of the number of messages.

S_a sends its join column values to site S_b. The number of messages received is determined by the size of the result of the semijoin $R_a <A=B] R_b$. If $J_s(R_a,R_b)$ is the selectivity of the semijoin, the number of tuples in the result relation will be $J_s(Ra,R_b)\cdot \mid R_b \mid$. In this example, $F_M(R_a.A) = 994$, so all join column values can fit in two message packets with control information, and $J_s(Ra,Rb)$ is 0.1, so the communications costs of the sequential semijoin methods are much lower than those of the other join methods. However, it is important to realize that the fractional communications cost (i.e., the communications cost as a portion of the total elapsed time) is not high in any of the join methods. For the non-semijoin methods, 100 messages were required in all, yielding a total message time of about 1.65 seconds.

The message cost analysis for the pipelined semijoin methods (PSNL and PSSM) is a bit more complicated. The number of messages sent by site S_a is

affected not only by the message packet size M, but also by the buffer space size B (in bytes) that is allocated at site S_a for holding R_a tuples. The *buffer factor*, F_B, is used to denote the number of R_a tuples which can be held in this buffer space:

$$F_B = \left\lfloor \frac{B}{tuple\_len(R)} \right\rfloor$$

To process the whole relation R_a, the number of "batches" (where a batch involves filling the buffer space with R_a tuples and processing these tuples) is given by $\left\lceil \frac{|R_a|}{F_B} \right\rceil$. The number of messages needed to send the join column values for one batch to site S_b can be expressed as $\left\lceil \frac{F_B}{F_M(R_a.A)} \right\rceil$. The total number of messages needed is thus the product of these two numbers, as is given in Table 3.2. In the example here, one buffer page is allocated for scanning R_a, and 22 tuples fit in a page, so $F_B = 22 << F_M(R_a.A)$. Thus, the number of tuples sent by one message is limited by the buffer space used, so the message packets going to S_b are not fully utilized. As a result, more messages are sent by S_a for the pipelined semijoin methods (PSSM and PSNL) than for the sequential semijoin methods. (Since one message packet can hold many join column values, it may be better in practice to select the number of R_a buffer pages according to the number of join column values that fit in one message packet.)

Another complication involved in the pipelined semijoin message analysis is that the number of messages received by site S_a is influenced by the distribution of the join column values. That is, each message sent to S_b results in at least one return message that either contains matching tuples or tells S_a to read

another page; the degree to which a given return message fills its message packet will depend on the number of matching R_b tuples found. For some messages sent by S_a, many values, hence more than one message packet, will be returned, but other messages may simply say "no matches" or may contain just a few tuples. As a result, the number of messages received by S_a will equal or exceed the number of messages sent by S_a, as shown in table 3.2. This is in contrast to the sequential semijoin case, which will totally fill all but the last of the message packets returned by S_b to S_a. For all of the reasons cited above, then, the number of messages for the PSSM and PSNL algorithms usually exceeds the number for SSSM and SSNL, as is the case in Figure 3.8. (Again, however, the total message cost is far from being the dominant cost factor here.)

Finally, for the semijoin-based sort-merge methods, fewer messages may be needed due to the differences in the ranges of the join column values of the two relations. One relation may be exhaust first when the other relation has only been partially scanned, in which case the corresponding data transfer is stopped. This probability is not reflected in Table 3.2.

The Number of Disk Accesses:

The number of disk accesses for a join method depends on the number of different pages accessed during the operation (of course), but it also depends strongly on the available buffer space, on the page replacement policy used by the buffer manager, and on the allocation of records to pages in each relation. Figure 3.9 shows the measured number of disk accesses at each site for the example query, and provides insight into the I/O cost of the various join methods. The differences in the number of disk accesses for the different join

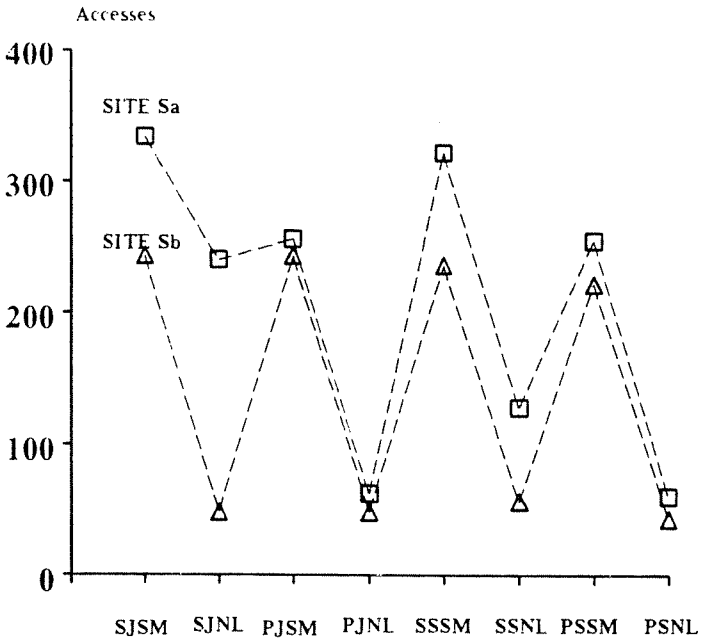

Figure 3.9: Number of disk accesses

methods can be explained to some extent by estimating the number of different pages accessed at each site for the different methods, although not every page access causes a disk access (since some pages might be already in the buffer).

Site S_b (where relation R_b resides) is examined first since its disk access analysis is simpler.

(1) SJNL and PJNL: At site S_b, both methods require just one scan of R_b to send it to S_a.

(2) SJSM and PJSM: Both methods require more page accesses in order to sort relation R_b.

(3) SSSM and PSSM: These two methods require the same number of page accesses to sort R_b. SSSM requires somewhat more page accesses because it has to store the intermediate result $R_a.A[A=B]R_b$ and then retrieve it again to send it back to S_a.

(4) SSNL and PSNL: the number of page accesses depends on the number of matching tuples in R_b and the depth of the index (i.e., the average number of data and index page accesses for each outer tuple from R_a). Like SSSM, SSNL requires more page accesses due to storing and retrieving the intermediate semijoin result.

The estimation process for the number of page accesses at site S_a is similar. The number of page accesses at S_a for the eight methods are as follows:

(1) SJSM and PJSM: Both methods have to sort R_a first. SJSM requires more page accesses for storing the remote relation R_b. At site S_a, the merge scan is done only on R_a for PJSM and on both R_a and the temporary R_b relation for SJSM.

(2) SJNL and PJNL: The number of page accesses for R_a is determined by the join selectivity. SJNL requires additional page accesses for storing the temporary R_b and building a B^+ tree index.

(3) SSSM and SSNL: Both methods require page accesses to scan R_a first to project the join column and eliminate duplicates. Then an index scan is used to fetch the tuples matching the tuples received from S_b for SSNL, while a merge scan is used for SSSM.

(4) PSSM and PSNL: For PSNL, the number of page accesses is simply the number of pages in relation R_a. PSSM requires more accesses to sort

relation R_a first before the merge scan.

3.4.2.2. Experiment 1: The Effects of Relation Size

Two groups of queries, QG1 and QG2, were tested to investigate the behavior of the different join methods as the relation sizes are varied. QG1 consists of joins between two relations of the same size; the result relations have the same sizes as the source relations (making the join selectivity simply the inverse of the source relation size). QG2 consists of joins between two relations of various differing sizes; the join selectivity is kept constant in query group QG2 (at a value of 10^{-4}). Since the two sites in the QG2 queries are asymmetrical, QG2 is further divided into two subgroups of queries, QG2.a and QG2.b. In QG2.a, the site having the larger relation was chosen as the join site; in QG2.b, the smaller relation resided at the join site. (As before, the result site is taken to be the join site for these tests.) These query groups are listed in Table 3.3.

Figure 3.10 shows the elapsed time measured for each of the QG1 queries. For the join of the two "tenthoustup" relations, the nested loops methods lost to the sort-merge methods even though the sort-merge methods must sort these

QG1			QG2					
			QG2.a			QG2.b		
$\|R_a\|$	$\|R_b\|$	|R|	$\|R_a\|$	$\|R_b\|$	|R|	$\|R_a\|$	$\|R_b\|$	|R|
10K	10K	10K	10K	1K	1K	1K	10K	1K
1K	1K	1K	10K	100	100	100	10K	100
500	500	500	10K	10	10	10	10K	10
100	100	100	10K	1	1	1	10K	1

Table 3.3: Sizes of relations used in query group QG1 and QG2

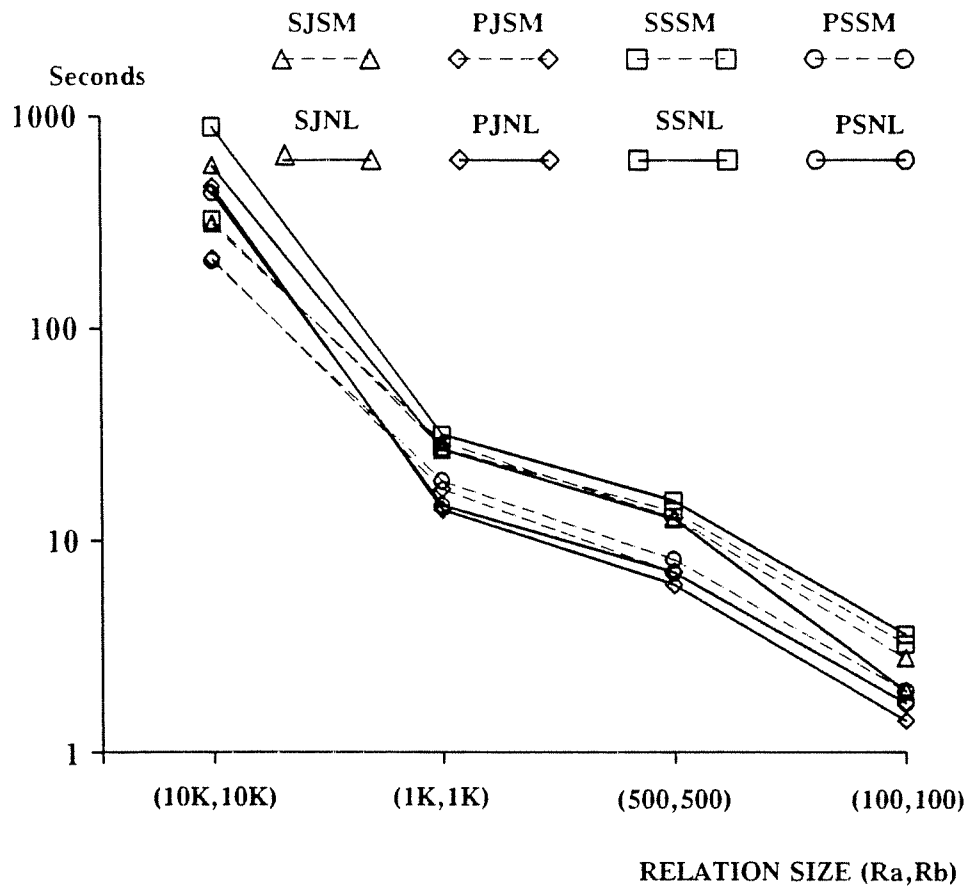

Figure 3.10: Elapsed time (QG1).

large relations. This is because the amount of work saved through sorting significantly outweighs the work required to perform the sorts. This is illustrated by Table 3.4, which shows the measured elapsed times and disk accesses for sorting the "tenthoustup", "thoustup" and "hundredtup" relations, and by the following analysis. It takes 64.89 seconds to sort the "tenthoustup" relation, and this involves 1911 disk accesses. This constitutes the pre-join "overhead" por-

Relation	Elapsed Time	Number of Disk Accesses		
		Total	Reads	Writes
hundredtup	0.54	13	6	7
thoustup	6.46	196	97	99
tenthoustup	64.89	1911	955	956

Table 3.4: Costs for sorting relations.

tion of the sort-merge methods for this case. After sorting, the merge phase accesses each page of each relation at most once.[†] In contrast, for the nested loops join using a nonclustered index, the number of disk accesses is much larger; this is due to the number of inner relation data pages (randomly) accessed. Figure 3.11 shows this clearly. Of the four nested loops methods, three of them required more than 10,000 disk accesses, which is what was chosen as an upper limit for the number of disk accesses traced due to space considerations. The one exception was PSNL (pipelined semijoin-based nested loops), which keeps tuples in memory at site S_a, scanning R_a only once. However, this method involved a large number of disk accesses at site S_b (not shown), where R_b is searched using the index to find the matching tuples for the 10,000 join attribute values sent by R_a. The elapsed time was mainly determined by the processing rate at site S_b in this case, which explains its elapsed time as compared to the sort-merge methods.

Table 3.4 also shows that, as the relation sizes decrease, the cost of sorting the relations begins to outweigh the cost of performing an inner relation disk access per outer relation tuple. With smaller relation sizes, Figure 3.11 shows that the total numbers of disk accesses for the pipelined nested loops methods

[†]The join column values are unique in this experiment.

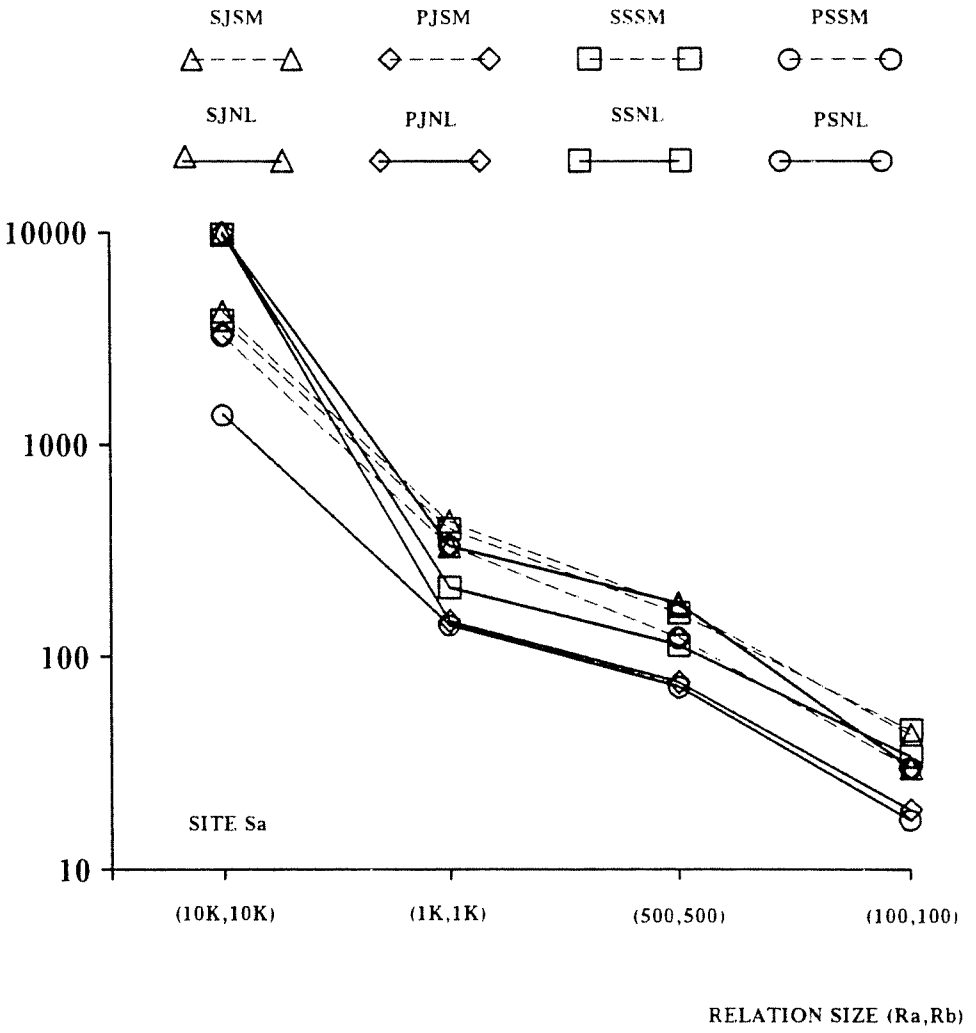

Figure 3.11: Number of disk accesses (QG1).

(PJNL and PSNL) are lower than those for the sort-merge methods. Thus, the pipelined nested loops methods are the best performers except at the largest relation size tested for QG1.

Figure 3.12 shows the total number of messages involved in executing each of the eight join methods tested. Only three curves are evident. The highest cost

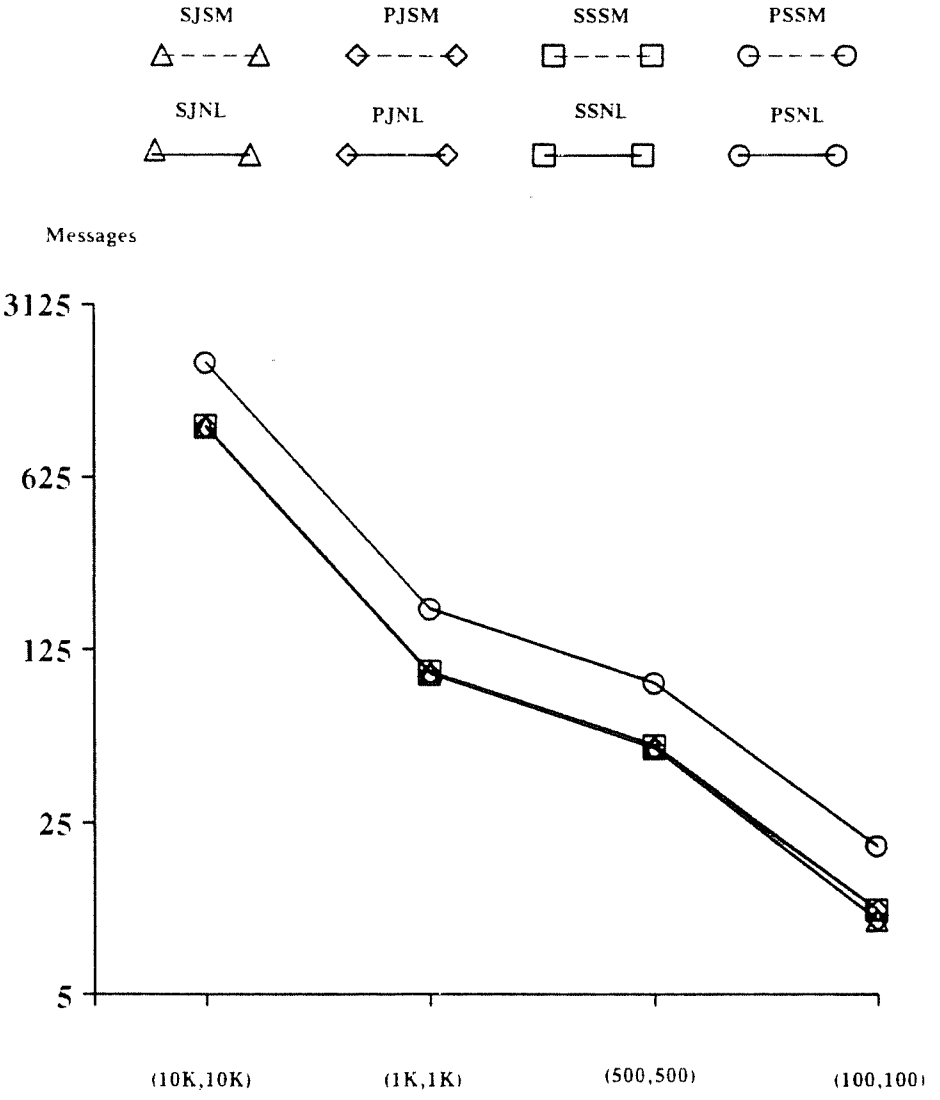

Figure 3.12: Number of messages (QG1).

here is for the pipelined semijoin methods, the next highest cost is for the sequential semijoin methods, and the lowest among the message costs are the four non-semijoin methods. This is because, in this case, the join is a "one-to-one join" — each tuple of R_a joins with exactly one tuple of R_b and vice versa. Thus, the use of semijoins here does not reduce the amount of data ultimately transferred to the join site; rather, it increases the overall message cost by the amount of data sent to the remote site from the join site initially. In all cases, given the packet transfer time of 16.6 milliseconds, the overall message time is less than 10% of the overall elapsed time. (Also, the effect of the message time is even less significant for the pipelined algorithms, as there is processing going on while messages are in transit.)

Figures 3.13 and 3.14 give the measured elapsed times for the queries in query groups QG2.a and QG2.b. These results clearly illustrate the differences between the various join methods tested. The diversity of the results can be explained based on the discussion of the detailed example and analysis given in the Section 3.4.2.1. It is evident from the figures that the nested loops join methods are more sensitive to relation size differences than the sort-merge methods, particularly at the larger relation sizes. This is because the sort-merge methods have a fixed component of their costs due to sorting the "tenthoustup" relation (see Table 3.4 for this cost). An extreme case is illustrated in Figure 3.14 for the the pipelined join case (PJSM). With the smaller relation site as the join site, PJSM's cost remains nearly constant over the whole size range investigated. The main components of the cost of PJSM are sorting R_b at site S_b and scanning R_b to send it to site S_a. (These two factors alone account for about 90% of the elapsed time.) The sort-merge methods can never execute fas-

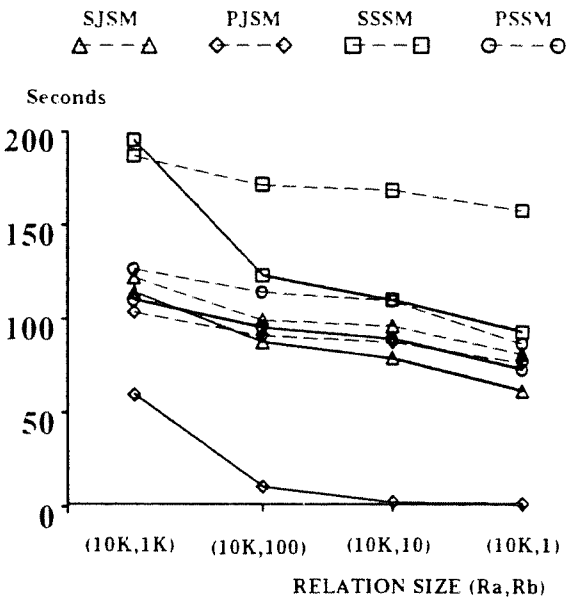

Figure 3.13: Elapsed time (QG2.a).

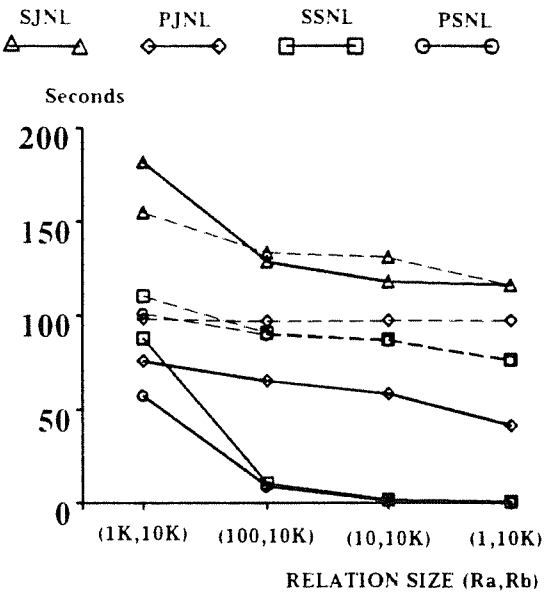

Figure 3.14: Elapsed time (QG2.b).

ter than the time it takes to sort and scan one of the two relations. The nested loops join methods are different, however. When the size of one of the source relations decreases, the number of disk accesses decreases dramatically for at least one of the nested loops methods in both query groups, as shown in Figures 3.15 and 3.16. This is due to the absence of sorting overhead and the effectiveness of the index for smaller outer relation sizes. The winner for query group QG2.a is the pipelined join version of nested loops (PJNL). The winner for query group QG2.b is the pipelined semijoin version of nested loops (PSNL); the sequential semijoin method (SSNL) is the next best choice, with nearly identical performance for the smaller relation sizes. While the message counts are not given here, they represent an insignificant portion of the overall query

processing cost (as in the previous cases examined).

One note here: It seems that the queries in QG2 are representative of a class of queries that is likely to arise in real database systems — that is, queries with a small number of tuples in one relation (the result of a selection) being joined with tuples from a much larger relation. An important observation from the tests covered by query groups QG1 and QG2 is that, when one relation is small, the pipelined nested loops join methods perform much better than their sequential counterparts or any of the sort-merge methods. When both relations are large, however, as when both were "tenthoustup" relations in the tests of QG1, the optimal methods will be the pipelined sort-merge methods.

3.4.2.3. Experiment 2: The Effects of Join Selectivity

Join selectivity, which is the ratio of the size of the result relation to the product of the sizes of the source relations, is an influential factor with respect to join algorithm performance. To see just how various join selectivities affect performance, tests on the eight distributed join methods using two relations each with 1000 tuples were run. However, these relations were not just "thoustup" relations from the Wisconsin database. Rather, the join selectivity was varied to obtain result sizes of 1000 tuples, 100 tuples, 10 tuples, and 1 tuple. Getting a result size of N tuples was accomplished by selecting R_a from the "tenthoustup" relation with "unique1" values in the range [4500..5499], selecting N of R_b's tuples from the "tenthoustup" relation randomly in the same range, and selecting the remainder of R_b's tuples randomly outside this range.

Figure 3.17 shows the measured elapsed times for the different join methods for the various join selectivities tested. Higher join selectivities (i.e.,

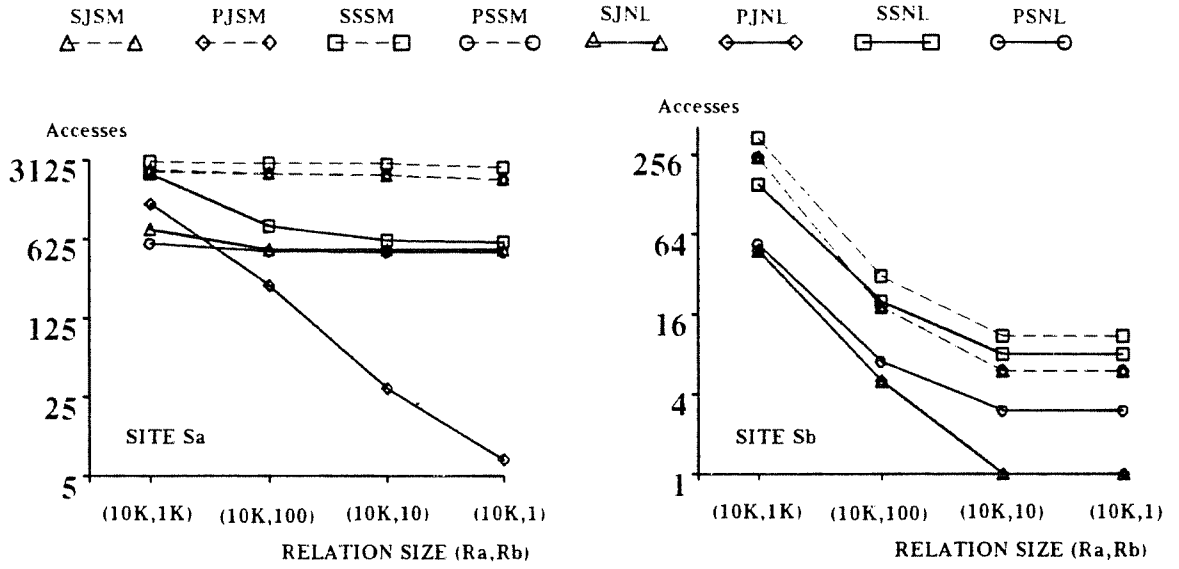

Figure 3.15: Number of disk accesses (QG2.a)

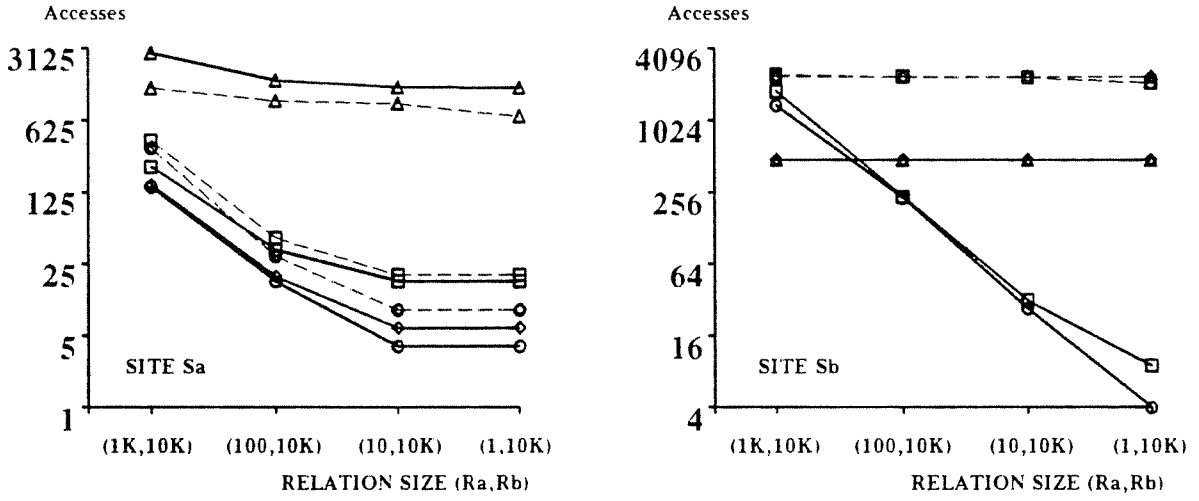

Figure 3.16: Number of disk accesses (QG2.b)

smaller result relations) mean that fewer tuples will match during the join, which leads to several cost savings. First, the result relation is smaller, so fewer disk accesses are needed to write out the result. Second, fewer data pages are accessed for the indexed nested loops join methods. Third, fewer tuples will be retrieved from the remote site for the semijoin methods, so the communications cost is reduced in their case. The pipelined nested loops join and semijoin methods were the winners in this experiment, with the semijoin method doing somewhat worse than the join-based method. Their pipelined sort-merge counterparts were the next best in terms of elapsed time here.

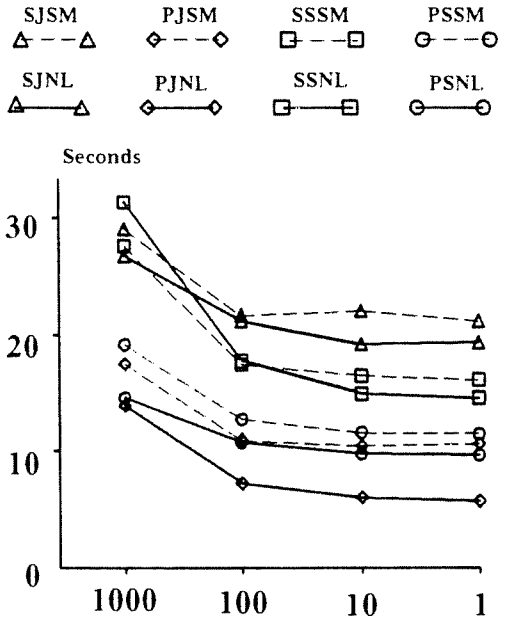

Figure 3.17:
Effects of join selectivity.

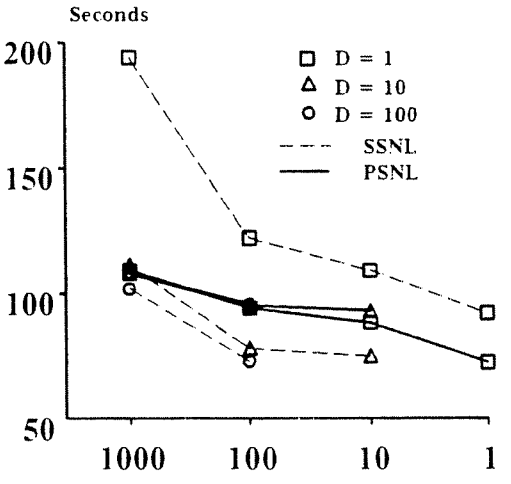

Figure 3.18:
Effects of duplicates.

3.4.2.4. Experiment 3: The Effects of Duplicate Attribute Values

Another factor which can influence the performance of a join method is the degree of join column value duplication. In the merge phase of a sort-merge join, duplicate join attribute values can cause multiple scans of pages of the inner relation. Perhaps more significant is the effect of duplicates on sequential versus pipelined semijoin performance. In the semijoin methods, site S_a sends the join column values of R_a to site S_b, and site S_b uses these values to fetch and return the matching tuples in relation R_b. In the sequential variants of the semijoin method, duplicate $R_a.A$ values are removed before sending them to S_b, which has two effects. First, less data is sent — duplicate join column values are avoided in messages from S_a to S_b, and (as a result) each matching R_b tuple is sent to S_a just once. The second (related) effect is the avoidance of multiple disk accesses in R_b for a given $R_a.A$ value. These two savings reduce both the communications cost and the local processing cost in comparison to the pipelined semijoin methods. A group of queries which join two relations on an attribute with duplicate values is tested in this experiment.

In order to quantitatively describe the degree of duplication, the *duplication factor D* of attribute A of relation R_a is defined. This factor is defined as the ratio of the number of tuples in relation R to the number of distinct values of attribute A. For example, in the "tenthoustup" relation shown back in Table 3.1, the duplication factor of the "hundred" attribute is 100, and that of the "thousand" attribute is 10. Figure 3.18 shows the effects of duplicates on the sequential and pipelined semijoin methods (SSNL and PSNL). In these tests, one of the source relations, R_a, had 10,000 tuples. The size of relation R_b was the same as the size of the result relation, so it was 1,000, 100, 10, and 1

(tuples) respectively. The join column of R_a, R_a.A, was varied so that different duplication factors could be obtained (1, 10, and 100). It can be seen from the figure that duplicates have almost no effects on PSNL. For SSNL, however, an increase in the duplication factor moves it from being much worse than PSNL to being much better. Obviously, the benefit due to the elimination of duplicates outweighs the usual disadvantages of the sequential methods when the duplication factor is high.

3.4.2.5. Summary of Test Results

There are several important observations to be made from the test results. First, if the issue of join site selection is set aside, Figures 3.13 and 3.14 show that the three nested loops methods PJNL, PSNL and SSNL provide the best performance when joining a large relation with a small one. For "medium" similar size relation joins, such as those shown in Figure 3.10, PJNL and PSNL also perform the best, and the sequential SSNL algorithm becomes worse than the two pipelined sort-merge methods (PJSM and PSSM). This leads to the conclusion that pipelined join and semijoin methods seem to be the most promising of the distributed join methods tested. To join very large relations of similar size, the pipelined sort-merge methods seem to be the best choices, as shown in Figure 3.10. These conclusions hold for the entire range of queries investigated.

Another general conclusion of this study is that the communications cost did not play a significant role in determining algorithm performance in the environment of this study. The contribution of communications costs to the overall measured elapsed times was around 10%. As an illustrative example, to

Relation Size	Elapsed Time (seconds)		
	local join	distributed join	degeneration (%)
|R| = |Rb| = 1K	55.8739	58.6128	4.90
|R| = |Rb| = 100	8.0877	9.3826	16.00
|R| = |Rb| = 10	0.9503	1.1164	17.48
|R| = |Rb| = 1	0.3500	0.3895	11.29

Table 3.5: A comparison of local and distributed joins (|Ra| = 10K).

transfer a "tenthoustup" relation from one site to another requires about 1000 messages, which only takes about 16-17 seconds. In queries where such a transfer might be useful, however, the processing cost may be as high as several hundred seconds. The relatively poor performance of the sequential semijoin methods provide additional evidence that a communications cost savings alone does not help much. As a more concrete example, Table 3.5 compares local join costs with distributed join costs in the same environment. In all cases, $|R_a|$ = 10K, and the local and distributed join methods used were those that gave the least elapsed time (i.e., local nested loops join with an index and PJNL). It is clear from the example that the message cost is not the major determining factor for performance.

The last point to be made is that choosing the right combination of a join processing site and a join method is important. Figures 3.13 and 3.14 indicate that, if the two relations to be joined have different sizes, the pipelined nested loops join method (PJNL) needs the site with larger relation to be the join site (i.e., it needs the outer relation to be the smaller of the two source relations). In contrast, the semijoin nested loops methods (PSNL and SSNL) perform much

better when the smaller relation site is chosen as the join site (i.e., they also need the outer relation to be the smaller of the two). The intuition behind these results is fairly simple, in retrospect: PJNL and PSNL are basically the same algorithm if the communications cost is zero, both being distributed executions of a simple nested loops join; they therefore have the same local processing costs. The outer relation should be the smaller of the two in the centralized case as well [Blas76][Seli79]. Note also that in a low communications cost environment such as this, the join's result site choice can be switched with little or no significant impact on performance. For example, suppose that R_a is a small relation, R_b is a large relation, and that S_a is to be the result site. The pipelined semijoin (PSNL) method is the best choice in this case. If S_b is the preferable result site for some reason, this can be arranged by using a pipelined join (PJNL) at site S_b instead.

3.5. SUMMARY

The performance of a number of different join methods for a distributed database system was studied in this chapter. Eight different methods were implemented on an experimental distributed computer system, the Crystal multicomputer. Join queries with various sizes, join selectivities, and attribute value distributions were tested. The results have shown that, in a local network, communications cost is not the dominant factor. Shipping an entire relation from one site to another site is a reasonable way to process a distributed join query — as long as it is done correctly. Correctly in this case means that a pipelined join algorithm, such as one where the outer relation is shipped to the join site (the inner site) in parallel with the local join processing itself, is employed. Although traditional (sequential) semijoin methods can reduce the

communications cost, and they perform well in cases where the join column duplication factor is high (i. e., many matching inner relation tuples per outer relation tuple), pipelined semijoin methods were found to be preferable in most of the test cases examined. These results hold over a wide range of query characteristics. For the case where two very large relations are to be joined, pipelined sort-merge methods are recommended. It was also found that the combination of the join method and the join site is important, as it is very important to ensure that the outer relation for the join is the smaller of the two source relations (as in centralized database systems).

The experiments described in this chapter are related to several other studies of distributed query processing techniques. First, the pipelined semijoin methods that were implemented are the ones used in System R^*, known there as the "fetch inner tuples as needed" methods [Seli80]. They opted to use the pipelined version of semijoin over the more traditional sequential version because they believed that it would tend to win in most situations due to lower local processing costs. The results obtained in this study indicate that this is indeed the case in a local network. The results of this chapter also concur with the claims of Page, which indicate that, in a distributed database system based on a local network, it is far more important that joins be done in the correct order and with the correct inner and outer relations than that they be done at the site which minimizes communications [Page83]. The key difference between Page's results and the results presented here is that his conclusions were based on a cost analysis of the INGRES database system and the LOCUS distributed operating system, whereas these results were obtained by measuring the performance of a number of actual distributed join queries. Finally, earlier analytical studies

have indicated that pipelined query evaluation techniques provide the best performance in centralized database systems [Smit75][Yao79]. The results of this chapter can be viewed as showing experimentally that pipelining is still the method of choice in a locally distributed database system (even when the pipeline is used to execute a single join operation).

CHAPTER 4

QUERY PROCESSING WITH LOAD BALANCING

This chapter presents a new approach to distributed query processing, *load-balanced query processing*, for locally distributed database systems. The distinguishing characteristic of this approach is that, in addition to picking a good static query processing plan, the subqueries are distributed dynamically in such a way as to balance the load of the system. Here, the phrase "locally distributed" refers to a collection of processing sites connected by a communications subnetwork with low latency and a high data transfer rate. In addition, uniform connectivity is assumed in this study. That is, the cost of transferring data from one site to another is assumed to be the same for all pairs of sites.

The first section briefly discusses the possible methods for integrating load balancing with distributed query processing. An approach based on static planning and dynamic allocation, Algorithm LBQP (**L**oad-**B**alanced **Q**uery **P**rocessing), is then described in detail. This algorithm consists of three phases — *static planning, dynamic allocation* and *refining*. Several heuristics are derived for the planning phase based on the experimental results presented in Chapter 3 and on other related research work. Heuristic algorithms are developed for the dynamic allocation phase and the performance of these algorithms is studied. The principles of the refining phase are discussed at the end of the chapter.

4.1. LOAD-BALANCED QUERY PROCESSING

The dynamic query allocation problem was studied for database systems with fully replicated data in Chapter 2. There are several ways to integrate the

basic ideas developed there with distributed query processing algorithms to achieve load-balanced query processing. *Interpretative planning and dynamic allocation, alternative compiled plans*, and *static planning and dynamic allocation* are three possible schemes.

4.1.1. Interpretative Planning and Dynamic Allocation

Interpretative query processing algorithms [Wong77][Epst78][Epst80b] decompose a user query into a sequence of subqueries (query pieces). The next subquery to be processed is selected at runtime after the previous one has been processed and its result relation has been obtained. The dynamic query allocation algorithms BNQ, BNQRD and LERT could be combined with such interpretative query optimization algorithms in either of two ways:

(1) The next subquery to be processed could be chosen as usual, i.e., without considering the current system load. Then, if there is more than one site where the chosen subquery can be processed, one of the dynamic query allocation algorithms (BNQ, BNQRD, or LERT) could be used to select the processing site.

(2) The load of different sites could be included in the processing costs for the subqueries by applying the LERT algorithm to each candidate subquery and processing site combination. The subquery and the site that lead to the least response time would be selected as the next processing step and site.

This approach to load-balanced query processing has several advantages. First, the exact sizes of the intermediate relations are known when query allocation decisions are made. The processing cost information needed for dynamic query allocation is therefore quite accurate. Second, the processing site for each

subquery is determined right before the subquery is to be executed, so the query allocation algorithm can use the most current system load information to obtain a better load balancing effect. However, this method has the standard drawbacks of all interpretative query optimization methods. First, it has a high runtime overhead since optimization takes place at runtime. In addition, this overhead is incurred every time a query is executed, even if it is to be executed many times.

4.1.2. Alternative Compiled Plans

User queries are precompiled into processing plans in some centralized and distributed database management systems [Seli80][Dani82]. If the storage sites and access paths of the relations referenced by a query remain unchanged after compilation, its processing plan is executed directly using its precompiled object modules. A possible approach to achieving load-balanced optimization when compilation is used would be to compile the query into several alternative plans instead of just one plan. Each plan would have an associated load constraint that specifies the load status under which it should be used as the processing plan for the query. Then, when the query is to be executed, the load status of the system is checked, and the plan with the load constraint closest to the current load is chosen as the processing plan. Alonso suggested a scheme like this for use in System R^* [Alon84].

Since precompilation is used in the alternative compiled plans approach, the query optimization overhead at runtime is fairly small. The only runtime overhead is the comparison of the current load with the load constraints of each of the alternative plans. However, the query optimization process itself becomes more complicated in this case, as both the possible load situations and the

storage sites of relations have to be considered at the same time. Considering more load-balanced alternatives will lead to more effective load balancing and hence better performance, but the number of alternative plans that can be considered is limited by storage space and by the complexity of the query plan itself. Furthermore, predicting possible system load conditions and choosing appropriate load constraints for the alternative processing plans may be quite difficult.

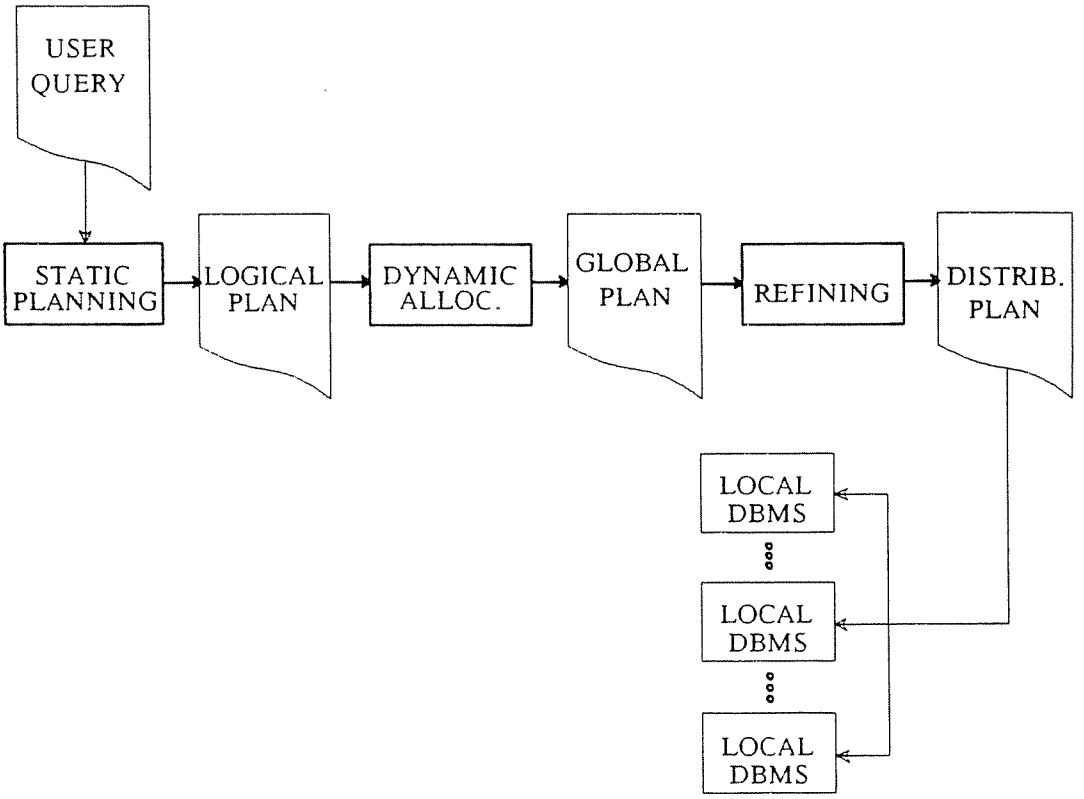

Figure 4.1: Algorithm LBQP: static planning plus dynamic allocation.

4.1.3. Static Planning and Dynamic Allocation

In the remainder of this chapter a third and different approach to the problem of load-balanced query processing is proposed. The proposed algorithm, Algorithm LBQP, uses the static planning and dynamic allocation scheme outlined in Figure 4.1. This approach represents a compromise between the two previously discussed possibilities. Query processing consists of three phases in Algorithm LBQP: the *static planning* phase, the *dynamic allocation* phase, and the *refining phase*. During the static planning phase, a user query is transformed into a *logical processing plan* which is a sequence of relational operations on logical relations (i.e., relations without physical locations having been bound yet). The local processing costs for each relation and the possible processing sites (the sites where the relation is stored) are attached to the plan for later use. The dynamic allocation phase performs the copy (site) selection process for the plan. In this phase, a dynamic allocation algorithm is applied to the logical plan, and a physical copy is selected for each relation referenced in the plan. The logical plan is thus transformed into a *global processing plan*. Finally, the refining phase chooses between semijoin-based and join-based pipelined distributed join methods for those joins that will be processed in a distributed fashion as a result of dynamic allocation. The refined global plan is then distributed to all participating sites. Each local DBMS translates its part of the plan into executable code and the query is executed.

This approach has two advantages. First, the separation of dynamic query allocation from static access planning simplifies the optimization process. Although some information must be recorded to facilitate the dynamic allocation phase, no new difficulties are introduced in the planning phase as compared

with existing query optimization algorithms. Existing algorithms can thus be easily augmented to include a dynamic allocation phase in order to achieve load balancing. Second, the major portion of the work required for query optimization, the generation of the logical plan, can be performed at query compilation time. Only the dynamic allocation and refining phases are done at runtime (right before the query is executed). Thus, the runtime overhead should be small.

It should be pointed out that the objective of load-balanced query optimization is to obtain an optimal plan under the current load conditions of the system. However, load balancing itself might introduce extra costs in the static sense. As an example from Chapter 2, a locally executable query might be shipped to a remote site for processing, and extra communications costs would thus be introduced. However, dynamic query allocation was still found to improve overall system performance in Chapter 2. The philosophy employed in Algorithm LBQP is to obtain a statically optimal local processing plan using some objective function, and then to execute the plan in the most favorable way given the current load status of the system. In the remainder of this chapter, the three phases of this approach are each described in turn.

4.2. THE STATIC PLANNING PHASE OF ALGORITHM LBQP

As mentioned above, statically optimal processing plans are generated for user queries in the static planning phase of Algorithm LBQP. Since the dynamic allocation and static planning phases are separate from each other, any existing query optimization algorithm could be used in the static planning phase to transform a query into a logical plan (as long as the information needed for

allocation is collected and attached to the plan). Therefore, it is not the purpose of this section to propose yet another detailed query optimization algorithm. Instead, the important principles for the static planning phase are discussed. The first part of the section discusses some heuristics that are applicable for static planning. Its most important conclusion is that issues related to the physical storage sites of the relations referenced by a query can be ignored during the static planning phase. The optimization method based on this approach and the structure of the resulting processing plan are then described.

4.2.1. Heuristics for LBQP

Based on the experimental results presented in the last chapter and other related research work, the following heuristic rules can be justified for the static planning phase in distributed database systems based on local area networks.

During this discussion, the two relations of a join are referred to as the outer and inner relations. The outer relation is the relation from which a tuple will be retrieved first, and the inner relation is the relation from which matching tuples will be retrieved for a given outer relation tuple. Using this convention, the processing cost for a join $R_a \rhd\lhd R_b$, where R_a is the outer relation and R_b the inner relation, can be expressed as:

$$TC(join) = AC(R_a) + AC(R_b \mid R_a) + CC(R_a, R_b) \qquad (4.2.1)$$

In equation (4.2.1), $TC(join)$ represents the *total cost* of processing the join, $AC(R_a)$ is the *access cost* of retrieving tuples from the outer relation R_a, and $AC(R_b \mid R_a)$ is the total access cost of fetching inner tuples from R_b with R_a as the outer relation. $CC(R_a, R_b)$ is the *communications cost* for transferring data between the sites of R_a and R_b during the join operation, which is zero in the

local join case. The cost of storing the result is not included in equation (4.2.1), as the result is assumed either to be transferred directly to the user terminal or to be used by the next operation in pipelined processing.

Heuristic 1. Only linear sequences of the possible of 2-way joins (i.e., joins of two relations with common join columns) are considered as candidates for the plan's join order for an n-way join $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$.

This is the same heuristic as that used in System R [Seli79] and System R^* [Seli80]. Linear join sequences have the form $((\ldots((R_1 \bowtie R_2) \bowtie R_3)\ldots) \bowtie R_n)$. While other join sequences could be considered, such as those of the form $(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4) \bowtie \ldots (R_{n-1} \bowtie R_n)$, considering only linear join sequences limits the search space considerably with the risk of missing the optimal sequence only in a few cases [Lohm85]. Furthermore, the resulting plan is easily executed in a pipelined way without requiring complicated synchronization. Intermediate relations are stored only when the next join uses the sort-merge algorithm and the sort operation is needed.

Heuristic 2. The optimal local join method for a join is unaffected by the locations (i.e., storage sites) of the participating relations. That is, if a join method (e.g., nested loops or sort-merge) is optimal for joining two relations at one site, its pipelined version will be the optimal method when the two relations are stored at different sites.[†]

As described in Chapter 3, the pipelined distributed join methods are extensions of the corresponding local join methods. The only difference is that tuples from the two relations to be joined are brought together via the communications

[†]Note: It is assumed here that all copies of a given relation have the same access paths available, an assumption that will be addressed in Section 4.2.2.

network in the distributed case. The communications cost for a distributed join is mainly determined by the sizes of the two relations rather than by the local join method. For example, both the pipelined nested loops join (PJNL) and pipelined sort-merge join (PJSM) algorithms transfer the entire outer relation. For pipelined semijoin methods, both the nested loops and sort-merge methods transfer the join column values of the outer relation and the matching tuples of the inner relation. Therefore, pipelined distributed join processing adds the same amount of communications cost to each of the local join methods, so the optimal method in the local case will still be optimal in the distributed case.

Heuristic 3. The optimal choice for the outer and inner relations of a join is unaffected by the location of the participating relations. That is, if the local join method requires R_a to be the outer (inner) relation for a join $R_a \bowtie R_b$, R_a should also be the outer (inner) relation when the join is processed in a distributed manner.

For the sort-merge join methods, the selection of the outer and inner relations is of little importance. As for the nested loops join methods, the experimental results for the distributed versions of these join methods in Chapter 3 suggest that the outer relation should be the smaller relation, just as in centralized systems.

One case where the choice for the outer and inner relations of a join might appear to be affected by the locations of the two relations is as follows. Consider the join $R_a \bowtie R_b$ where R_a is smaller than R_b and is therefore chosen as the outer relation in the local case. When the two relations are at different sites and the desired result site is the site of R_a, the optimal distributed join method might at first seem to be to use R_b as the outer relation to eliminate the

communications cost for transferring the result back to the site of R_a. That is, R_b would be sent to the site of R_a, and the join would be processed there. In Chapter 3, however, it was shown that pipelined semijoin methods and pipelined join methods have similar performance but different result sites. Thus, instead of switching the outer and inner relations, a pipelined semijoin method can be used in place of a pipelined join method, leaving the choice for the outer and inner relations the same as in the local case. (As will be seen later, this switch is actually considered in the refining phase of algorithm LBQP.)

Heuristic 4. The optimal join sequence (order) chosen for an m-way local join remains optimal or nearly optimal if part of the sequence is processed in a distributed manner.

This heuristic is based on the following basic observations: (1) Different join orders usually result in local processing costs with large differences, and (2) communications costs in local area networks are a secondary component of the total processing cost (as shown in Chapter 3). Consider the following 3-way join as an example:

retrieve into R $(R_a.all, R_b.all, R_c.all)$

where $R_a.A = R_b.A$ and $R_c.C = R_a.C$

Let us denote this join by $R_a \bowtie R_b \bowtie R_c$, and let $((R_a \bowtie R_b) \bowtie R_c)$ be the optimal join order when all relations are at the same site as in Figure 4.2(a). Further, suppose that each join is to be processed using PJNL (the pipelined nested loops join method).

If the relations are at three different sites as in Figure 4.2(b), it can be shown that it is quite likely that $(R_a \bowtie R_b) \bowtie R_c$ will still be the optimal order. In case (b), the total processing cost for this locally optimal join order, $(R_a \bowtie R_b) \bowtie R_c$ is:

$$TC = AC_{total} + CC_{total} \tag{4.2.2}$$

where AC_{total} is the total local access cost and CC_{total} is the total communications cost. These two cost components can be expressed as:

$$AC_{total} = AC(R_a) + AC(R_b \mid R_a) + AC(R_c \mid R_{ab}) \tag{4.2.3}$$

and

$$CC_{total} = CC(R_a, R_b) + CC(R_{ab}, R_c) + CC_b(R) \tag{4.2.4}$$

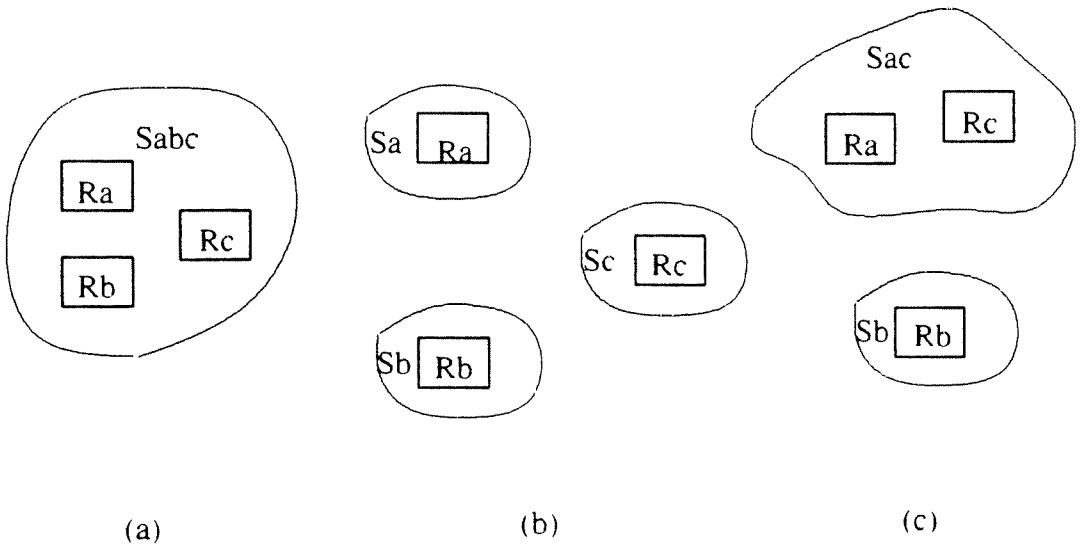

Figure 4.2: Different physical locations of three relations.

where R_{ab} is the result relation of the join $R_a \bowtie R_b$, and $CC(R)$ is the communications cost for sending the final result relation R to the result site.

Consider another join order, say, $(R_a \bowtie R_c) \bowtie R_b$. Its total processing cost is

$$TC' = AC'_{total} + CC'_{total} \tag{4.2.5}$$

with

$$AC'_{total} = AC(R_a) + AC(R_c \mid R_a) + AC(R_b \mid R_{ac}) \tag{4.2.6}$$

$$CC'_{total} = CC(R_a, R_c) + CC(R_{ac}, R_b) + CC'(R) \tag{4.2.7}$$

where R_{ac} is the intermediate result relation of the join $R_a \bowtie R_c$. As shown in Chapter 3, communications costs are secondary in local area networks, so:

$$AC_{total} >> CC_{total} \tag{4.2.8}$$

and:

$$AC'_{total} >> CC'_{total} \tag{4.2.9}$$

The difference between the processing costs of the two join orders is:

$$TC' - TC = AC_{diff} + CC_{diff} \tag{4.2.10}$$

where:

$$AC_{diff} = AC'_{total} - AC_{total}$$

$$CC_{diff} = CC'_{total} - CC_{total}$$

Since $(R_a \bowtie R_b) \bowtie R_c$ is the optimal order in the local case, and different join orders usually result in local processing cost with large differences, it is expected that $AC_{diff} >> 0$. It is also expected that $AC_{diff} >> CC_{diff}$ since communications costs are secondary. Thus, $TC' - TC > 0$, and the locally

optimal join order will still be optimal.

Even if the communications cost difference CC_{diff} were large enough to be a factor in determining the value of $TC' - TC$, there is little chance that $CC_{diff} << 0$. From Equations (4.2.4) and (4.2.7), the difference in the communications costs for these two join orders is:

$$CC_{diff} = (CC(R_a,R_c) + CC(R_{ac},R_b) + CC'(R)) - (CC(R_a,R_b) + CC(R_{ab},R_c) + CC(R))$$

Since the pipelined nested loops join method is to be used for the joins, both $CC(R_a,R_c)$ and $CC(R_a,R_b)$ are simply the cost of transferring relation R_a to a remote site. That is:

$$CC(R_a,R_c) = CC(R_a,R_b)$$

and thus:

$$CC_{diff} = (CC(R_{ac},R_b) - CC(R_{ab},R_c)) + (CC'(R) - CC(R))$$

Since $(R_a \bowtie R_b) \bowtie R_c$ is the optimal order in the local case, the intermediate relation R_{ab} is expected to be smaller than the result relation of $R_a \bowtie R_c$, R_{ac}. Thus:

$$CC(R_{ac},R_b) > CC(R_{ab},R_c)$$

As a result, the only thing that could make $CC_{diff} << 0$ would be if $CC'(R) - CC(R) << 0$. This could only be true if S_b is the result site and thus $C'(R)$ is zero (since otherwise the two communications costs for transferring the result relation are the same). However, the discussion in Heuristic 3 about switching between pipelined join methods and pipelined semijoin methods is fully applicable in this case. The second join in $(R_a \bowtie R_b) \bowtie R_c$, the join

$R_{ab} \bowtie R_c$, can be processed using the pipelined semijoin method (PSNL). This would make S_b the result site, making $CC(R)$ zero as well. Note that this will change the cost $CC(R_{ab},R_c)$ for the join $R_{ab} \bowtie R_c$, but the change will most probably decrease this cost since semijoin methods tend to transfer less data (see Chapter 3). Thus, the locally optimal join order will be still optimal.

Figure 4.2(c) shows another possible distribution of the three relations. In this case, R_a and R_c are at the same site and R_b is at another site. A possible optimal order with this distribution when S_b is the result site is $(R_a \bowtie R_c) \bowtie R_b$, since the original optimal order involves more data transfers (from S_{ac} to S_b for the first join, from S_b to S_{ac} for the second join, and then from S_{ac} back to S_b for returning the results). As discussed in case (b), however, the local processing cost is the dominant factor in the total processing cost, and the local processing cost of the join $R_a \bowtie R_c$ is likely to be much greater than that of the join $R_a \bowtie R_b$. Furthermore, the communications cost $CC(R_{ac},R_b)$ should be greater than $CC(R_{ab},R_c)$, and the additional communications cost for returning the results can again be virtually eliminated by using the pipelined semijoin method to process the second join. The locally optimal join order should still be optimal in this case as well.

Heuristics 2 through 5 described above lead to an important conclusion in regard to the static planning phase of the load-balanced query processing algorithm: *For locally distributed database systems, an optimal or near optimal processing plan that specifies the join order, the local join methods and the right outer and inner relations can be obtained by ignoring the physical storage sites of the relations in the system.* A locally distributed database can thus be treated as a centralized database in the static planning phase.

4.2.2. Static Optimization and the Logical Plan Structure

Logically, a relational database is a set of relations, denoted by $R_1, R_2, \ldots R_n$. Physically, each relation R_i has one or more copies, denoted by $R_{i1}, R_{i2} \ldots R_{ik}$. Each copy is stored at a *site*. When a user query references relation R_i, the database management system will perform operations on one or more copies of R_i. In order to distinguish a relation in the abstract sense from its physically stored copies, the term *logical relation* is used to refer to the relation in the abstract sense, and the physically stored copies are referred to as *physical relations*. By ignoring storage site issues during static planning, optimization is actually performed using the logical relations referenced by a query.

One issue related to using logical relations has to do with the access paths that are associated with each physical copy of a logical relation. So far, all copies of a relation have been assumed to be identical and to have the same access paths available. These access paths are used during optimization. This assumption is reasonable for locally distributed databases, where relations will most likely have identical copies. However, if different access paths are available for different copies of a relation, the most favorable access path available will be used for static optimization purposes. Later on, when the dynamic allocation phase considers which copy of each relation to use, the cost of building any missing access paths can be considered as their various copies are examined.

The overall objective of Algorithm LBQP is to minimize the total processing cost, including CPU, I/O, and communications costs. Since static planning is performed by viewing a locally distributed database as a centralized one, the objective in static planning is thus to minimize the local processing cost

components (the CPU and I/O costs). An access path selector such as that of System R [Seli79] could be used to generate the logical plan. The System R optimizer works as follows: First, all possible pairs of relations with common join column attributes are considered. For each of these two-way joins, the join method and access paths with the minimum processing cost are retained. Join methods and access paths are then found for joining a third relation with the results of each possible two-way joins. At every processing step, then, a k^{th} relation is joined with the results of joining the previous k-1 relations. After all of these plans have been obtained, the one with the minimum processing cost is chosen as the final processing plan.

The output of the static planning phase of Algorithm LBQP is a logical plan that specifies the following information in the form of a processing graph:

(1) The join order for all joins in the query.

(2) The local join method for each join.

(3) The outer and inner relation for each join.

(4) The access paths and predicates that will be used in accessing each relation.

(5) The information needed by the dynamic allocation phase, including the storage sites of each relation, the estimated CPU cost and I/O cost for accessing each relation, and the estimated sizes of the intermediate and result relations. (In cases where the access paths for different physical copies of a relation are not identical, the costs of building the required access path must also be recorded.)

The processing graph for a logical plan is a tree. The leaves of the tree are relation nodes and the internal nodes are operation nodes. The structure of

```
operation_node = record
                   operation : {select, project, NL_join, SM_join, ...};
                   outer_relation : pointer;
                   inner_relation : pointer;
                   result_size : integer;
                   result_site : integer;
                 end;

relation_node = record
                   relation_name : string;
                   feasible_assignment_set : set of site;
                   access_path : (index_type, attr, op, value);
                   access_predicate : {(attr, op, value) ...}
                   io_time : time;
                   cpu_time : time;
                   data_len_join : integer;
                   data_len_semijoin : integer;
                 end;
```

Figure 4.3: Structure of the nodes in the processing graph

these nodes is shown in Figure 4.3. An operation node holds the information about an operation, including the type of operation, its operands and an estimate of the size of the result relation. The join methods specified in an operation node are local join methods since the system is viewed as a centralized system in the static planning phase. In the refining phase, these operations will be further specified as being either semijoin-based or join-based if the source relations are at different sites. The designation of the outer and inner relations is done in the static planning phase; for monadic operations, the inner field is simply set to nil. Finally the *result_size* field indicates the estimated size for the intermediate

result relation of the operation (used later for estimating the subsequent communications cost).

A relation node holds the information about a relation on which an operation is to be performed. This includes the name of the relation, its storage site(s), information about the access path used by the operation, and its processing cost. The access path information includes the type of access path, the index attribute specification(if any), and the remaining conjunctive predicate to be applied to accessed tuples (if any). In the case where different physical copies have different access paths, the cost of building the required access path would also be included in the relation node. The processing cost for a relation is estimated during static planning and expressed as a *cpu_time* and an *io_time*. The fields *data_len_join* and *data_len_semijoin* contain the data (in bytes) that would be transferred in distributed join and semijoin operations, which are used later in the refining phase of LBQP (discussed in Section 4.4).

For the example query and database shown in Figure 4.4, the processing graph for the static plan would have the form shown in Figure 4.5.

4.3. THE DYNAMIC ALLOCATION PHASE OF ALGORITHM LBQP

The output of the static planning phase is a *logical plan* that references *logical relations*. The function of the dynamic allocation phase of Algorithm LBQP is to map these logical relations to their physically stored copies. In this phase, which takes place at runtime, a query is viewed as a sequence of *query units*. The dynamic query unit allocation algorithm developed in this section is applied to the sequence, and processing sites are determined for each of the query units in the sequence.

(a) A sample university database and some statistics:

Relation	Cardinality	Key-Cardinality	Tuple Length
DEPT (DID, DNAME, CHAIRMAN)	50	DID-50	62
STUDENT(SID, SNAME, MAJOR)	40K	SID-40K	42
COURSE (CNO, CNAME, OFFERED_BY)	2K	CNO-2K	34
ENROLLED (SID, CNO, GRADE)	150K	SID-35K CNO-1.5K	16

(b) Access paths available:

Relation	Clustered Index	Non-Clustered Indexes
DEPT	DID	---
STUDENT	SID	MAJOR
COURSE	CNO	OFFERED_BY
ENROLLED	SID	CNO

(c) An example query written in QUEL:

"for all students who major in Computer Sciences, print their ID numbers and names, the name of the courses they are enrolled in, and their grades in these courses"

```
range of D is DEPT
range of S is STUDENT
range of C is COURSE
range of E is ENROLLED
retrieve (S.SID, S.SNAME, C.CNAME, E.GRADE)
   where D.DNAME = "Computer Sciences"
      and S.MAJOR = D.DID
      and E.SID = S.SID
      and E.CNO = C.CNO
```

(d) The distribution of the relations:

Site S_1	Site S_2	Site S_3	Site S_4
DEPT	DEPT	STUDENT	COURSE
COURSE	STUDENT	ENROLLED	ENROLLED

Figure 4.4: An example query for a distributed university database.

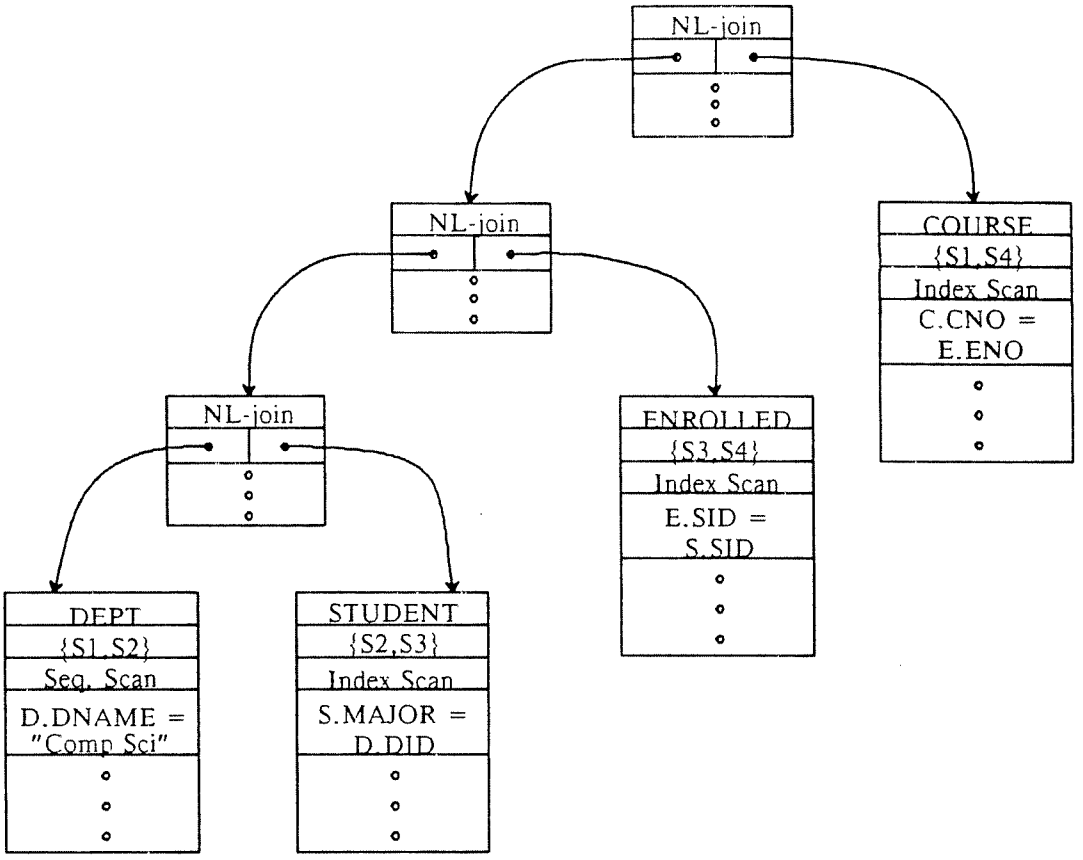

Figure 4.5: Processing graph for the query in example 4.4.

This section first discusses the criteria for dynamic query unit allocation. BNQ-based and BNQRD-based query unit allocation algorithms are then developed. The plans generated by these heuristic algorithms are compared with those of an exhaustive search algorithm to study the optimality of the heuristic algorithms. The execution time of these algorithms is also analyzed.

4.3.1. Load Unbalance Factor

Since the main goal of dynamic allocation is to achieve a load balanced system, a quantitative description of the "balancedness" of a system is needed. For this purpose, the *load unbalance factor* proposed by Livny [Livn83] is extended in this section.

Recall that in algorithm BNQ, the load of a site s_j is expressed as the number of query units currently at that site. That is,

$$LD_{BNQ}(s_j) = NQ(s_j)$$

Under this definition of load, Livny's load unbalance factor, ***UBF***, can be expressed as the maximum value of the load differences between the n sites as follows:

$$UBF_{BNQ} = \max_{1 \le j,k \le n} | LD_{BNQ}(s_j) - LD_{BNQ}(s_k) |$$

However, given that query units have constraints as to where they can be executed (i.e., only at sites holding the necessary data), this measurement is not sufficient to serve as the load unbalance measurement of the system here. For example, if the possible execution sites for every query unit of a query happen to exclude the sites with the initial maximum and minimum load in the system, the *UBF* defined above will remain constant regardless of the assignments of query units to sites. Using the concept of *variance*, the unbalance factor can instead be defined as the variance of the load distribution in the system. That is,

$$UBF_{BNQ} = \frac{\sum_{j=1}^{n} (LD_{BNQ}(s_j) - \overline{LD}_{BNQ})^2}{n}$$

$$= \frac{\sum_{j=1}^{n} (NQ(s_j) - \overline{NQ})^2}{n}$$

where $NQ(s_j)$ is the number of queries at site s_j and $\overline{NQ}$ is the average number of query units per site after all query units are allocated.

The BNQRD algorithm classifies query units into two classes, I/O-bound and CPU-bound. The load of a site s_j is then expressed as the number of I/O-bound and CPU-bound query units at s_j:

$$LD_{BNQRD}(s_j) = \{ NQ_{IO}(s_j), NQ_{CPU}(s_j) \}$$

where NQ_{IO} and NQ_{CPU} are the number of I/O bound query units and the number of CPU bound query units respectively. Load balancing in algorithm BNQRD is performed on a per-class basis, so the unbalance factor for BNQRD-based allocation can thus be defined as:

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

The last dynamic query allocation algorithm studied in Chapter 2 was the LERT algorithm. As indicated by the simulation results, both BNQRD and LERT outperform the BNQ algorithm. The simulation results also showed that BNQRD performs as well as LERT, except in a few cases where the result fraction, i.e., the result's communications cost is relatively large. However, BNQRD requires less information about the query units and is simpler than LERT, so this study concentrates on a BNQRD-based dynamic query unit allocation algorithm.

4.3.2. Query Units in the Logical Plan

From the viewpoint of dynamic allocation, a query is simply a linear sequence of processes. Each process requires a certain amount of service from the system's resources, and adjacent processes may communicate with each other. The query plan can thus be viewed as a list of query units for use in the dynamic allocation phase, where a query unit is defined as the maximum processing unit that only references one relation in the processing graph (a relation node in the graph). Typical examples of a query unit are: accessing tuples from the outer relation in a join, or fetching matching tuples from the inner relation given the outer relation tuples and then merging these tuples to produce the result relation. (In general, a query unit may be an arbitrarily complex single relation subquery, similar to a one-variable query in INGRES [Wong76].)

The logical plan obtained in the static planning phase, which is an input to this second phase, contains all the information about the query units needed by the dynamic allocation phase. The estimated CPU and I/O time will be used to determine whether a query unit is I/O-bound or CPU-bound in the BNQRD-based allocation algorithm. In Algorithm LBQP, a query unit is always assigned to a site where a physical copy of the relation referenced by the query unit is available. The storage sites of the relation referenced by the query unit form the *feasible assignment set* for the query unit. In the BNQ- and BNQRD-based allocation algorithms, the communications cost between two query units is considered to be either 0 if the two adjacent query units are allocated to the same site, or 1 if these two query units are allocated to different sites. Therefore, the result size information in the processing graph is not used in this phase (but it will be used in the refining phase).

The originating site and result site of a query are also part of the input of the dynamic allocation phase. To account for the communications cost of sending a description of the query to a site other than its originating site, a "dummy" query unit is used to represent the query initiation. Its feasible assignment set contains only one site, the query's originating site. There is no processing cost associated with this query unit. However, if the processing site of the first query unit is not the same as the originating site, the communications cost will count towards the total communications cost. Similarly, another "dummy" query unit with zero processing cost is used to represent the result site of the query in order to account for the communications cost of sending the results back to the result site.

4.3.3. The Query Unit Allocation Problem

Based on the concept of query units and the definition of the unbalance factor, the query unit allocation problem in the dynamic allocation phase can be stated as follows.

The information given is:

(1) A locally distributed relational database with n sites, $\{s_1, \ldots, s_n\}$;

(2) A user query Q expressed as a sequence of query units, Q = $\{q_1, \ldots, q_m\}$, which is to be processed in a pipelined manner.

(3) A feasible assignment set $S_i = \{s_{i_1}, \ldots, s_{i_k}\}$ for each unit q_i, $1 \leq i \leq m$, specifying where copies of the relation referenced by q_i are stored.

(4) The communications cost C_i between each pair of query units q_i and q_{i+1} assuming that q_i and q_{i+1} execute at different sites. (If q_i and q_{i+1} are

allocated to the same site, C_i will end up being 0.) For $0 < i < m$, C_i is the transmission cost for intermediate results. C_0 represents the cost of initiating a query at a site other than its originating site, and C_m is the cost of sending the result of the query to the specified result site.

(5) The initial load at each site s_j, $1 \leq j \leq n$, expressed as $LD_{BNQ}(s_j)$ for BNQ-based allocation or $LD_{BNQRD}(s_j)$ for BNQRD-based allocation.

The problem is to find an optimal allocation plan[†] OPT such that:

(1) The unbalance factor under this plan, UBF^{OPT}, is minimized, i.e. $UBF^{OPT} = \min_{a \in A} UBF^a$, where a is an allocation plan, and A is the set of all possible allocation plans.

(2) The total communications cost, $C_{comm} = \sum_{i=0}^{m} C_i$, is minimized.

Here, minimizing the communications cost is considered as one of the objectives since the logical plan obtained in the static planning phase was optimized with regard to the total local processing cost (including I/O cost and CPU cost) only. However, since local processing costs, and therefore load balancing, play a more important role than communications cost with regard to the performance of a locally distributed database system, the optimal allocation plan will be considered to be the one with the minimum total communications cost among those plans with the (same) minimum *UBF* value.

This allocation problem has several unique and important features when compared with the previous research work on task allocation discussed in Section 1.2. First, the main objective here is load balancing, and the

[†]The term 'plan' here and in entire Section 4.2 simply refers to an allocation plan of query units to processing sites, unless otherwise specified.

communications cost is considered to be a secondary consideration. Most other task allocation algorithms, even those claiming to consider load balancing, have other primary objective functions such as inter-process communications cost [Chu80] [Ma82]. Second, the load balancing requirement is quantitatively defined via the load unbalance factor *UBF*, and it is hence as computable as other objective functions. Third, the initial system load is taken into account, and allocation is performed right before the execution of the query.

Since the objective of load balancing is quantitatively well-defined, the dynamic query unit allocation problem, with its objectives of minimizing the unbalance factor of the system and then the communications cost, can be formalized (see Appendix A). However, there are several difficulties involved in trying to apply the non-heuristic solution methods reviewed in Section 1.2. Aside from computational complexity when the number of sites becomes large, a difficulty in trying to use an integer programming approach is that the two objective functions, minimizing the unbalance factor and the communications cost, are in conflict. They therefore cannot simply be added together and treated as a single function. The branch-and-bound search method cannot be used in this case either, as the main evaluation function, the unbalance factor *UBF*, cannot be computed during the expansion of the search tree, as assigning a query unit to a site may either increase or decrease the *UBF*. Thus, the unbalance factor of an allocation plan is computable only after all query units have been assigned to sites. That is, all possible allocation plans would end up being generated, as in exhaustive search methods.

Perhaps a more important consideration is that the dynamic allocation phase is performed at runtime, so it should introduce as little overhead as possible.

Heuristic methods requiring less computational effort and providing near-optimality are therefore preferable. In the following sections, heuristic algorithms for solving the dynamic query unit allocation problem are presented.

4.3.4. The Basic BNQ-Based Algorithm

Since the BNQ-based allocation algorithm is the simplest one, it will be described first. Figure 4.6 gives the algorithm. The input of the algorithm includes a *feasible assignment set* for each query unit and an *initial load vector.* This vector specifies the number of existing processes (i.e., query units) at each site when the query are to be allocated. The originating site and result site of the query are also given as input.

As shown in figure 4.6, query units are allocated one by one without backtracking or reassignment during allocation. Therefore, the order in which the query units are allocated to sites is a critical factor affecting the optimality of the resulting plan. The concept of the *degree of freedom* of a query unit is introduced to control the allocation order. Two such metrics are used by the algorithm, and their computation is shown in figure 4.7. The first is the "static" degree of freedom, given by the number of sites where a query unit can be allocated (i.e., the cardinality of its feasible assignment set). If the static freedom of a query unit is one, there is no freedom at all — that is, the query unit must be allocated to the only possible site. If the static freedom is greater than one, there is more than one candidate site. A query unit with a higher static degree of freedom will be relatively more flexible than one with a lower static degree of freedom due to more site choices.

```
Algorithm BNQ_Load_Balanced_Allocation

input : num_qus : integer;        (* number of query units *)
        num_sites : integer;      (* number of processing sites *)
        originating_site, result_site : site;
        initial_load : array [1..num_sites] of integer;
        feasible_assignment_sets : array [1..num_qus] of set of site;

output: allocation_site : array [0..num_qus+1] of site;

var qu_no : integer;  (* the id of a query unit *)

  (* DegreeOfFreedom(qu_no) computes static and dynamic freedom of query
       unit qu_no;
     Update(site_no) updates the current load vector and the freedom for
       query units because a query unit is assigned to site site_no;
     NextQueryUnit returns the qu_no of the next query unit to be allocated,
       which is the query unit with the smallest static degree of freedom
       (or NIL if there are no more query units to be allocated);  *)

begin
  allocation_site[0] := originating_site;
  allocation_site[num_qus+1] := result_site;
  for qu_no := 1 to num_qus do
    DegreeOfFreedom(qu_no);
  end;
  qu_no := NextQueryUnit;
  repeat
      allocation_site[qu_no] := SelectSite(qu_no);
      Update(allocation_site[qu_no]);
      qu_no := NextQueryUnit;
  until qu_no = NIL;
end.
```

Figure 4.6: BNQ-based heuristic allocation algorithm.

```
procedure DegreeOfFreedom(qu_no : integer);
var
  site_no : integer;
begin
    static_freedom[qu_no] := 0;
    dynamic_freedom[qu_no] := 0;
    for site_no := 1 to num_sites do
       if s in feasible_assignment_sets[qu_no] then begin
            static_freedom[qu_no] := static_freedom[qu_no]+1;
            dynamic_freedom[qu_no] := dynamic_freedom[qu_no]
                        + current_load[site_no]);
       end;
    end;
end;
```

Figure 4.7: Procedure computing freedom.

The other kind of freedom has a more "dynamic" flavor. The dynamic freedom of a query unit is the sum of the current load of the sites in its feasible assignment set. (The current load of a site is the sum of its initial load and the number of query units which have been assigned to it thus far.) When two query units have the same static freedom value, the query unit whose feasible assignment sites are more heavily loaded will usually have fewer choices.

One heuristic used to reduce the amount of computation required for allocation is that sites with a current load larger than the average load after allocation are considered to be full. Such sites will not be assigned further query units (except if such a site is the only site in the feasible assignment set for a query unit). Whenever a site is full, either due to the initial load or to the assignment of a new query unit to the site, it is deleted from the feasible assignment set of all query units. Note that this changes their static and dynamic

degrees of freedom, so these must be updated throughout the allocation process.

The query units in a query are allocated one by one in order of their degree of freedom. That is, query units with fewer site choices are allocated earlier. The degree of static freedom is the primary consideration; query units with the same degree of static freedom are ordered according to their degree of dynamic freedom. (Note that these freedom measures are somewhat different — a larger static freedom value implies more site choices, whereas a larger dynamic freedom value implies fewer site choices.)

In order to select a site for a query unit from its feasible assignment set, the *current load* and *potential load* of the candidate sites and the *benefit* and *potential benefit* of possible assignments are evaluated. The *current load* of a site is the total number of query units at that site when a query unit is to be allocated, including both the initial load at that site and any query units already assigned to it during allocation. *The potential load* of a site s_j is the load that might be introduced by those unassigned query units q_j that have s_j in their feasible assignment set. *The benefit* of an assignment of a query unit q_i to s_j is the communications cost savings due to this assignment. If q_{i-1} (or q_{i+1}) is already allocated to s_j, and q_i is allocated to the same site s_j, the communications cost between q_{i-1} (or q_{i+1}) and q_i will become zero. If neither q_{i-1} or q_{i+1} has been allocated to site s_j, then the benefit of the assignment is zero. However, if s_j is in the feasible assignment set of q_{i-1} (or q_{i+1}), and this query unit has not been allocated to any site yet, there is still a possibility that the communications cost between q_i and q_{i-1} (or q_{i+1}) could become zero. This would happen if q_i is first assigned to s_j, and then q_{i-1} (or q_{i+1}) is later assigned to the same site. Compared with assigning q_i to a site s_k that is not in the feasible

```
(* This function returns the benefit of allocating qu to site site_no *)
function Benefit(qu : integer; site_no : integer) : integer;
var benefit : integer;
begin
    benefit := 0;
    if (allocation_site[qu + 1] = site_no) then
      benefit := benefit + 1;
    if (allocation_site[qu - 1] = site_no) then
      benefit := benefit + 1;
    Benefit := benefit;
end;

(* This function returns the potential benefit of allocating
   query unit qu to site site_no *)
function PotentialBenefit(qu : integer; site_no) : integer;
var pot_benefit : integer;
begin
    pot_benefit := 0;
    next_qu := qu + 1;
    while (next_qu <= num_qus) and UnAssigned(next_qu)
          and (site_no in feasible_assignment_set[next_qu]) do begin
      pot_benefit := pot_benefit + 1;
      next_qu := nex_qu + 1;
    end;
    next_qu := qu - 1;
    while (next_qu >= 1) and UnAssigned(next_qu)
          and (site_no in feasible_assignment_set[next_qu]) do begin
      pot_benefit := pot_benefit + 1;
      next_qu := nex_qu - 1;
    end;
    PotentialBenefit := pot_benefit;
end;
```

Figure 4.8: Functions for computing benefit and potential benefit.

assignment set of q_{i-1} or q_{i+1}, then, assigning q_i to s_j has the potential of being beneficial. This possible communications cost savings is defined as the *potential benefit* of assigning q_i to s_j.

These four metrics — current load, potential load, benefit and potential benefit — are used to evaluate the assignment of query unit q_i to site s_j, and a site is selected for q_i after comparing the values of these parameters among all sites in the feasible assignment set of q_i. Figure 4.8 shows the functions that compute the benefit and potential benefit of an assignment of a query unit to a site.

Since load balancing is the main objective, the evaluation order for these four metrics during the allocation process is: minimum current load, maximum benefit, minimum potential load and maximum potential benefit. First, an attempt is made to allocate query unit q_i to the site with the minimum current load among its feasible assignment sites. If more than one site has the same minimum load, the benefit of assigning q_i to each site in the set of minimum load sites is calculated, and the site with the maximum benefit is selected. If the number of sites providing the maximum benefit is also greater than one, then the site with the minimum potential load among them is chosen. Finally, the maximum potential benefit site is considered if there is more than one site with the same minimum potential load. Figure 4.9 shows the SelectSite function, illustrating this screening process. It is important to notice that not all four metrics are calculated for every query unit. For most query units, only one or two metrics will need to be evaluated. Furthermore, the size of the candidate site set gets smaller and smaller as each metric is considered in turn. This is advantageous for minimizing the cost of the site selection process.

```
function SelectSite(qu_no: integer) : site;
var allocate_site : site;

(* NumMinLoadSites(qu_no) finds the sites with the minimum
     load in the feasible assignment set of query unit qu_no, and
     return the number of such sites.

   NumMaxBenefitSites(qu_no) finds the sites with the maximum
     load in the feasible assignment set of query unit qu, and
     return the number of such sites.

   NumMinPotentialLoadSites(qu_no) finds the sites with the
     minimum potential load in the feasible assignment set of query
     unit qu, and return the number of such sites.
*)

begin
  if NumMinLoadSites(qu_no) = 1
  then allocate_site := min_load_site[qu_no]
  else if NumMaxBenefitSites(qu_no) = 1
      then allocate_site :=max_benefit_site[qu_no]
      else if NumMinPotentialLoadSites(qu_no) = 1
          then allocate_site := min_potential_load_site[qu_no]
          else allocate_site := any site in {max_potential_benefit_sites};
  SelectSite := allocate_site;
end.
```

Figure 4.9: Function SelectSite.

Figure 4.10 shows an example input for the BNQ-based dynamic query unit allocation algorithm. The example query consists of 5 query units, and there are 8 sites in the system. The algorithm is applied as follows:

i. The total initial load is 12, so the expected balanced load is $\left\lceil \frac{(12+5)}{8} \right\rceil = 3$. Since site s_1 is full initially, it is deleted from the feasible assignment sets of q_1 and q_4.

ii. The static degree of freedom for the query units, including dummy units q_0 and q_6, is {1, 3, 3, 4, 1, 5, 1}, so q_0 and q_6 are allocated to s_3 first.

iii. q_4 is allocated to s_6, the only site in its feasible assignment set.

iv. Both q_1 and q_2 have the same static degree of freedom value of 3. The feasible assignment set of q_1 is $\{s_2, s_5, s_7\}$, with the relevant part of the current load vector being {1, 2, 1}, and q_2's is $\{s_2, s_4,$

An Example Input:

number of query units	5;
number of sites in the database	8;
query originating site	s_3;
query result site	s_3;
initial load vector	{4, 1, 1, 2, 2, 0, 1, 1};
feasible assignment set	
q_1	{1, 2, 5, 7};
q_2	{2, 4, 5};
q_3	{3, 5, 7, 8};
q_4	{1, 6};
q_5	{3, 4, 5, 6, 7};

Figure 4.10 : An example for query allocation.

$s_5\}$, with relevant part of the current load vector $\{1, 2, 2\}$. Thus, q_2 has less dynamic freedom, so q_2 is allocated next.

v. q_2 is allocated to site s_2, as s_2 is the site with the minimum load (of 1) in its feasible assignment set. (This increases s_2 's load by 1.)

vi. q_1 is now allocated to site s_7 for the same reason as in v.

vii. Since q_3 is next in increasing order of static freedom, it is considered next. Its relevant part of the current load vector and feasible assignment set are $\{1, 2, 2, 1\}$ and $\{s_3, s_5, s_7, s_8\}$, respectively. The minimum load sites for q_3 are $\{s_3, s_8\}$. As for benefits, benefit(q_3,s_3) = benefit(q_3,s_8) = 0. Finally, potential_load(s_3) = 2, but potential_load(s_8) = 1. q_3 is thus allocated to site s_8.

viii. Finally, q_5 is considered: Its feasible assignment set is $\{s_3, s_4, s_5, s_6, s_7\}$, and its corresponding load vector is $\{1, 2, 2, 1, 2\}$. s_3 and s_6 have the same current load of 1, but benefit(q_5, s_3) = 0 and benefit(q_5, s_6) = 1 (since q_4 was already allocated to s_6 in (iii)). Thus, q_5 is allocated to site s_6.

ix. The final assignment is:

query unit	(0)	1	2	3	4	5	(6)
execution site	(3)	7	2	8	6	6	(3)

After assignment, the load vector is {4, 2, 1, 2, 2, 2, 2, 2}. The ***UBF*** for this assignment is 0.61, and its total communications cost is 5. Comparing this result with the optimal plan, obtained by the exhaustive search method described in the next section, it turns out that this is actually an optimal allocation plan under the definition.

4.3.5. A Study of the Optimality of the BNQ-Based Algorithm

The algorithm just described is a greedy algorithm in the sense that query units are assigned one by one, without looking ahead or backtracking. However, the potential load and potential benefit metrics are used to improve its optimality. Also, the query unit allocation order and the order in which the four metrics are considered was carefully designed. In order to evaluate the optimality of this heuristic algorithm, a study of how the query unit allocations obtained using the algorithm compare to the corresponding optimal allocations was conducted. Figure 4.11 depicts the testing process. A query generator was used to generate queries with an initial load vector and a feasible assignment set for each query unit in the queries. Both the exhaustive search program and the BNQ-based heuristic algorithm were then applied to these queries. (The exhaustive

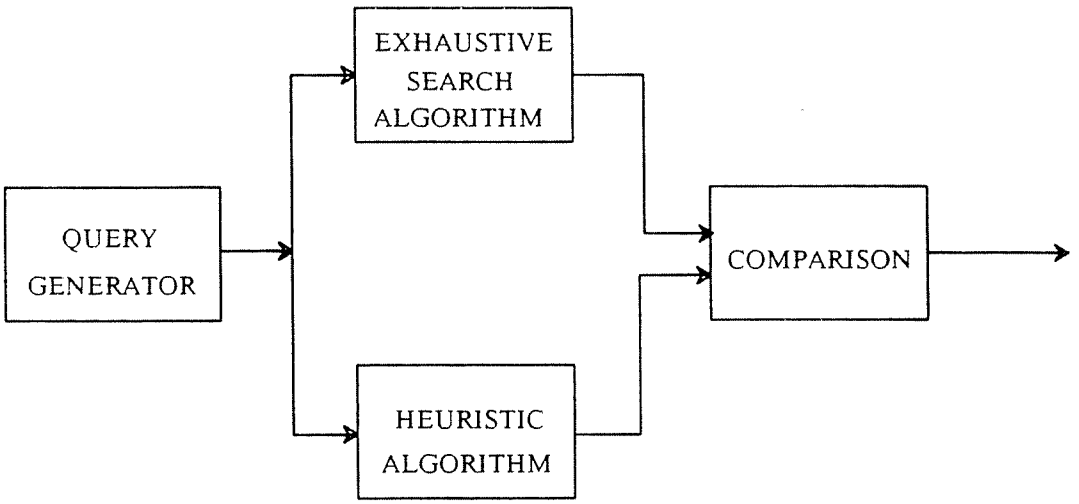

Figure 4.11 : Testing the optimality of the heuristic algorithm.

search method used always finds an optimal plan for a given input.) The heuristic plans were then tested for optimality by comparing them with the optimal ones. The optimality of the heuristic algorithm can be characterized by the percentage of heuristic plans which have the same *UBF* values and total communications costs as the optimal plans for some number of tests.

The parameters that controlled the query generator were the number of query units in the query, the number of sites in the system, a maximum load parameter and the average number of copies of each relation referenced by the query. (It is assumed that each query unit references a different relation.) The maximum load parameter controls the initial unbalance of the system, as the initial load of each site is assumed to be uniformly distributed between zero and this maximum value. The average number of copies was used to determine the size of the feasible assignment set of each query unit. That is, for each query unit and site, the probability that the relation referenced by the query unit has a copy at the site is the average number of copies divided by the number of sites in the system. By changing this parameter, the extent of replication of the database can be varied from a fully replicated database to a partially or non-replicated database.

Optimal allocation plans were obtained using an exhaustive search. In order to obtain the optimal plan for a given query, a search tree was constructed according to the feasible assignment sets of the query units comprising the query. Figure 4.12 shows the search tree for the example query of Figure 4.10. Each level of the tree corresponds to the possible allocations of one of the query units, with each node at a level representing one site in the feasible assignment set of the query unit. (The dummy query units q_0 and q_6 are not shown since

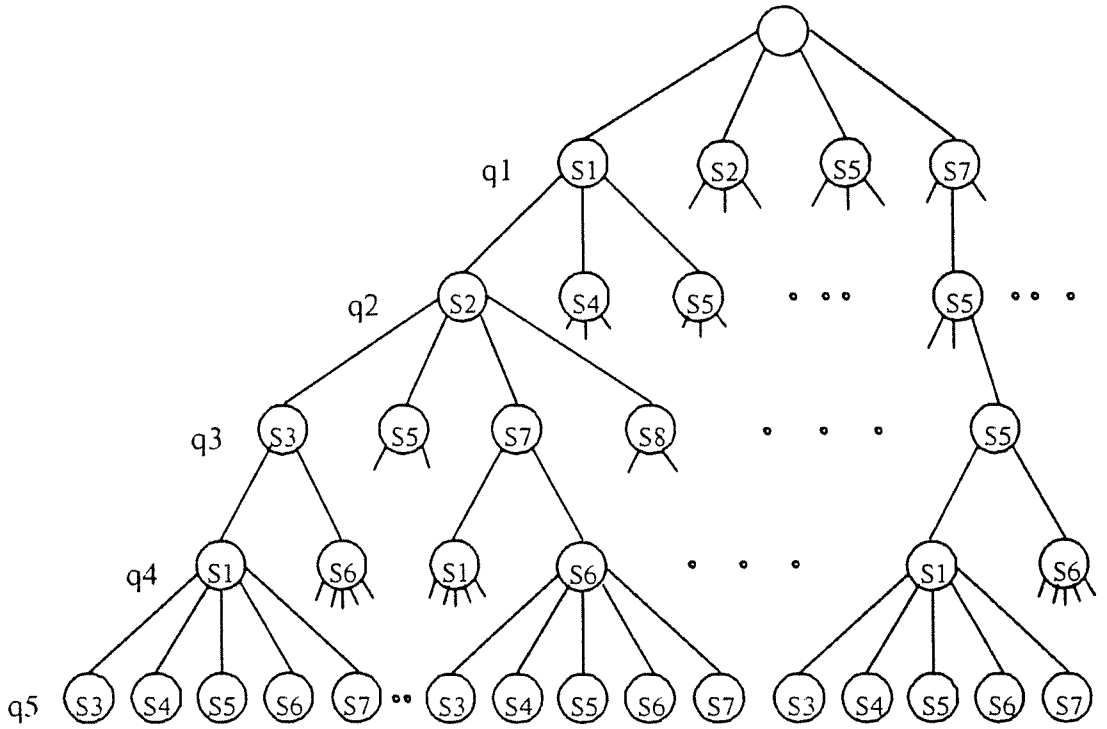

Figure 4.12 : A search tree for finding the optimal plan.

each has only one possible allocation site.) This tree is evaluated after all possible assignment sites have been considered by computing the unbalance factor and the communications cost along each branch of the tree.

The following tests were conducted in order to investigate the optimality of the heuristic algorithm for different queries and different system configurations:

(1) In the first group of tests, the number of query units (m) of a query was varied from 3 to 6. The number of sites in the database (n) was varied from 4 to 12. The maximum load was randomly chosen in the range 4 to 12,

and the average number of copies was set to the half the number of sites.

(2) The second group of tests fixed the number of query units (m) at 5 and the number of sites (n) at 8. The average number of copies was set to 4 for this group of tests. The maximum initial load was varied to investigate the performance of the heuristic algorithm when the initial unbalance factor was changed.

(3) In the third group of tests, the number of copies was varied. The number of query units (m) and sites (n) were fixed at 5 and 8, respectively. The average number of copies was set to 8 (fully replicated), 4 (each relation is stored at half of the sites) and 2 (only two copies of each relation are stored).

The test results are summarized in Tables 4.1 to 4.3. In these tables, the percentage of optimal plans obtained by using the heuristic algorithm is given for each test. Also, the percentage of plans that are optimal under each of the objective functions alone is given. (These plans either have the same unbalance factor as the optimal plan but a larger communications cost, or have the same or lower communications costs but a larger unbalance factor). The tables indicate that, in most cases, the heuristic algorithm generated the optimal plan. The percentage of optimal plans is at least 75%, and typically higher, for the runs in Test 1. About 95% of the plans obtained in Test 1 are optimal if only the unbalance factor is considered, which is encouraging since load balancing is the primary objective. As for the communications costs, more than 80% of the plans have the same (or lower) values as the optimal plan.

As the number of sites and query units is increased, however, the number of the optimal plans found by the heuristic algorithm decreases. The explanation

Optimality of the Heuristic Algorithm (Test 1)					
m	n	total runs	Percentage of the Optimal Plans (%)		
			optimal plans	optimal on UBF	optimal on CC
3	4	100	97	100	97
	6		98	100	98
	8		93	98	95
	10		99	99	100
	12		97	99	98
4	4	100	91	100	91
	6		94	100	94
	8		85	96	88
	10		90	97	93
	12		94	98	96
5	4	100	86	100	86
	6		88	99	89
	8		85	98	87
	10		80	98	82
	12		79	95	84
6	4	100	75	100	75
	6		81	97	83
	8		76	93	83
	10		70	92	78
	12		77	92	84

Table 4.1: Optimality in general (Test 1).

Optimality of the Heuristic Algorithm (Test 2, m = 5, n = 8)				
max. init. load	total runs	Percentage of the Optimal Plans (%)		
		optimal plans	optimal on UBF	optimal on CC
2	100	79	98	81
8		85	98	87
16		93	100	93

Table 4.2: Optimality versus the initial load (Test 2).

Optimality of the Heuristic Algorithm (Test 3, m = 5, n = 8)				
avg. num. copies	total runs	Percentage of the Optimal Plans (%)		
		optimal plans	optimal on UBF	optimal on CC
2	100	97	98	99
4		85	98	87
8		26	100	26

Table 4.3: Optimality versus the number of copies (Test 3).

for this is that larger numbers of sites and query units increase the number of possible plans, so the probability of selecting an optimal plan using a greedy algorithm decreases. It is then more likely that a nearly optimal plan will be chosen instead of the true optimum. This is especially clear in the fully replicated case, where the heuristically obtained plans frequently result in higher communications costs. In this case, a query unit can be allocated to any site in the system. The exhaustive search method can make use of this flexibility to decrease the communications cost, but this is much less true for the heuristic algorithm. On the contrary, having more choices reduced the chances of the heuristic algorithm generating an optimal plan. Two enhancements are proposed to improve the heuristic algorithm in this respect.

4.3.6. Enhancing the BNQ-Based Algorithm

Two enhancements have been designed to further minimize the communications cost, and hence to improve the overall optimality of the resulting allocation of query units to sites. Both enhancements start from the plan obtained using the basic algorithm. Local adjustments are then made to the plan to decrease its communications cost while keeping the unbalance factor constant.

Enhancement 1. Enhancement 1 is applied to each query unit individually. If a query unit q_i was assigned to site s_j, its feasible assignment set is searched to find another site s_k so that, if q_i is assigned to s_k, the unbalance factor of the system is not affected but the communications cost decreases. Figure 4.13 shows an example of this enhancement. After applying the BNQ-base dynamic query unit allocation algorithm, q_3 is allocated to site s_3. In this case, s_6 is in the feasible assignment set of q_3, and the initial load of both s_3 and s_6 is 0. Apply-

Input :

number of query units	3;
number of sites	6;
query originating site	s_5;
query result site	s_5;
initial load vector	{2, 5, 0, 4, 5, 0};
feasible assignment set	
q_1	{1, 3, 4};
q_2	{2, 6};
q_3	{2, 3, 5, 6};

allocation sites after applying BNQ-based algorithm:

query unit	(0)	1	2	3	(4)
allocated site	(5)	3	6	3	(5)

allocation sites after applying Enhancement 1:

query unit	(0)	1	2	3	(4)
allocated site	(5)	3	6	6	(5)

Figure 4.13: An example of Enhancement 1.

ing Enhancement 1 will reallocate q_3 to s_6, which will not affect the unbalance factor but which will reduce the communications cost. Tables 4.4 to 4.6 are the results for the three tests with Enhancement 1 applied following the heuristic algorithm. It can be seen that there is an improvement in terms of the communications cost. In the partially replicated case, the number of the optimal plans increased by 2-10%, while in the fully replicated case, the number of optimal

Optimality of the Heuristic Algorithm (Test 1)					
m	n	total runs	Percentage of the Optimal Plans (%)		
			optimal plans	optimal on UBF	optimal on CC
3	4	100	98	100	98
	6		98	100	98
	8		97	98	99
	10		99	99	100
	12		98	99	99
4	4	100	93	100	93
	6		96	100	96
	8		90	96	93
	10		95	97	98
	12		96	98	98
5	4	100	92	100	92
	6		94	99	95
	8		88	98	90
	10		88	98	90
	12		83	95	88
6	4	100	84	100	84
	6		84	98	86
	8		78	93	85
	10		78	92	86
	12		79	92	86

Table 4.4: Optimality in general (Test 1, Enhancement 1).

Optimality of the Heuristic Algorithm (Test 2, m = 5, n = 8)				
max. init. load	total runs	Percentage of the Optimal Plans (%)		
		optimal plans	optimal on UBF	optimal on CC
2	100	84	98	86
8		88	98	90
16		98	100	98

Table 4.5: Optimality versus the initial load (Test 2, Enhancement 1).

Optimality of the Heuristic Algorithm (Test 3, m = 5, n = 8)				
avg. num. copies	total runs	Percentage of the Optimal Plans (%)		
		optimal plans	optimal on UBF	optimal on CC
2	100	98	98	100
4		94	98	96
8		43	100	43

Table 4.6: Optimality versus the number of copies (Test 3, Enhancement 1).

plans nearly doubled.

Enhancement 2. Enhancement 1 adjusts the allocation site for each query unit individually from within its feasible assignment set. Enhancement 2 takes a more global view. The main idea of this enhancement is to group as many query units as possible together at the same site without affecting the ***UBF***. This enhancement looks at each pair of adjacent query units (q_i, q_{i+1}), where q_i and q_{i+1} have been allocated to two different sites s_i and s_k, and tries to find a third query unit $q_j (j > i+1)$ which is also allocated at site s_i. If sites s_i and s_k are in the feasible assignment sets of q_{i+1} and q_j respectively, then it is possible to reverse the choice of sites for q_{i+1} and q_j without affecting the ***UBF*** (since the number of query units at s_k and s_j will not change). However, this switch eliminates the communications cost between q_i and q_{i+1}. For each query q_i, $1 \leq i \leq m-2$, for which q_{i+1} is at a different site, Enhancement 2 looks for such a q_j. Figure 4.14 shows an example of this enhancement. The allocation plan for $q_0 - q_5$ before applying Enhancement 2 is {5, 2, 5, 5, 2, 5}. It is obvious that reversing the allocation sites for q_2 and q_4 will not change the ***UBF*** value, but this will decrease the overall communications cost.

Tables 4.7 to 4.9 shows the test results repeated with both enhancements employed. The results show that the optimality of the algorithm is further improved, and that Enhancement 2 is especially helpful in the fully replicated case (where the optimal plan is now always found). The optimality of the heuristic approach proposed here has been shown to be good. While the test results may vary with different input data, it is expected that the general trends will remain the same.

Input :

number of query units 4;
number of sites 6;
query originating site s_5;
query result site s_5;
initial load vector {4, 1, 4, 5, 1, 3};

feasible assignment set
q_1 {1, 2, 3, 6};
q_2 {1, 2, 5, 6};
q_3 {3, 5};
q_4 {2, 3, 5, 6};

allocation sites after applying BNQ-based algorithm:

query unit	(0)	1	2	3	4	(5)
allocated site	(5)	2	5	5	2	(5)

allocation sites after applying Enhancement 2:

query unit	(0)	1	2	3	4	(5)
allocated site	(5)	2	2	5	5	(5)

Figure 4.14: An example of Enhancement 2.

4.3.7. The Cost of the BNQ-Based Algorithm

Besides the optimality of allocations generated by the heuristic algorithm, cost is another important concern. As mentioned earlier, dynamic allocation takes place at runtime, so its cost should be as small as possible. A test was conducted to measure the CPU time of the algorithm and its enhancements. In the test, the number of sites was fixed at 8. The number of copies of referenced relations was varied to simulate both fully replicated and partially replicated

Optimality of the Heuristic Algorithm (Test 1)					
m	n	total runs	Percentage of the Optimal Plans (%)		
			optimal plans	optimal on UBF	optimal on CC
3	4	100	100	100	100
	6		100	100	100
	8		97	98	99
	10		99	99	100
	12		99	99	100
4	4	100	96	100	96
	6		98	100	98
	8		93	96	96
	10		96	97	99
	12		97	98	99
5	4	100	94	100	94
	6		97	99	98
	8		94	98	96
	10		91	98	93
	12		88	95	93
6	4	100	90	100	90
	6		90	98	92
	8		81	93	88
	10		81	92	89
	12		84	92	92

Table 4.7: Optimality in general (Test 1, Enhancements 1 & 2).

Optimality of the Heuristic Algorithm (Test 2, m = 5, n = 8)				
max. init. load	total runs	Percentage of the Optimal Plans (%)		
		optimal plans	optimal on UBF	optimal on CC
2	100	85	98	87
8		94	98	96
16		99	100	99

Table 4.8: Optimality versus the initial load (Test 2, Enhancements 1 & 2).

Optimality of the Heuristic Algorithm (Test 3, m = 5, n = 8)				
avg. num. copies	total runs	Percentage of the Optimal Plans (%)		
		optimal plans	optimal on UBF	optimal on CC
2	100	97	98	99
4		94	98	96
8		100	100	100

Table 4.9: Optimality versus the number of copies (Test 3, Enhancements 1 & 2).

systems. The test was run on a VAX 11/780, and CPU times were obtained using functions provided by UNIX. The results are shown in Table 4.10. It can be seen that the execution times are fairly small. Even with a query consisting of 5 query units and a fully replicated system with 8 sites, the total CPU time is less than 40 milliseconds.

Another cost consideration is how execution time increases when the number of query units and the number of sites is increased. The complexity of the basic algorithm can be estimated as follows: Before selecting allocation sites

Execution Time of the Basic Allocation Algorithm (in msec)

number of query units	average number of copies			
	2	4	6	8
3	17.0	21.0	22.2	24.7
4	23.9	26.1	28.4	31.3
5	27.1	33.4	34.6	40.9

Execution Time of Enhancements 1 (in msec)

number of query units	number of multiple copies			
	2	4	6	8
3	2.1	2.3	2.7	3.0
4	2.2	1.9	3.1	4.6
5	2.0	2.7	3.4	3.7

Execution Time of Enhancements 2 (in msec)

number of query units	number of multiple copies			
	2	4	6	8
3	1.2	0.3	0.8	0.7
4	0.7	1.1	0.8	0.7
5	1.0	0.6	1.0	0.8

Table 4.10: The execution time of the allocation algorithm.

for the query units, the degree of freedom of each query unit is calculated. The function *NextQueryUnit* is then called to find the query unit to be allocated next, and *SelectSite* is called to select a processing site for this query unit. After a site has been selected, the system load vector and the degree of freedom of the remaining query units are updated. This process is repeated until all query units have been allocated. Table 4.11 summarizes the complexity of this dynamic query unit allocation procedure. Since every query unit is processed in this way, the total complexity of the algorithm is $O(max(mn, m^2 \log_2 m))$.

Note that, since m is small for most realistic queries, and n is really the number of sites in the union of the feasible assignment sets for the subqueries (which may be smaller than the number of sites in the system), this seems quite acceptable. In order to see how the algorithm's execution times compare to those of the exhaustive search, the elapsed time of both algorithms were measured during the tests presented in the last section. The results are shown in Figure 4.15. These tests were performed on a VAX 11/750 running UNIX with just one user on the system. The figures show that the elapsed time of the heuristic algorithm is indeed basically linear in n when m is fixed, but that the

Complexity of the BNQ-based Allocation Algorithm	
Function or Procedure	Complexity
DegreeOfFreedom(qu_no)	$O(n)$
NextQueryUnit	$O(m \log_2 m)$
SelectSite(qu_no)	$O(n)$
Update(allocation_site[qu_no])	$O(m)$
TOTAL	$O(max(n, m \log_2 m))$

Table 4.11: Complexity analysis per query unit.

exhaustive

search is not. When the number of queries and sites are small, the heuristic algorithm has only a small advantage with regard to execution time. However, when the number of query units and sites increases, the elapsed time of the exhaustive search increases dramatically. For example, when $m=3$ and $n=4$, its elapsed time is 52.1 milliseconds. For $m=6$ and $n=12$, it becomes 16.2 seconds, or over 300 times longer. In contrast, the execution time of the heuristic algorithm is basically linear in mn, going from 47.3 milliseconds to 243 milliseconds for these same values of m and n.

Another observation from figure 4.15 is that the enhancements do increase the computation time somewhat. Thus, applying the enhancement procedures should perhaps be optional. In most cases the basic algorithm performs satisfactorily and it is not necessary to apply the enhancements. In some cases, such as in a fully replicated system or when the number of sites is large, either or both of the enhancement procedures can be applied.

4.3.8. A BNQRD-Based Version of the Algorithm

As indicated in Chapter 2, the information-based BNQRD algorithm will outperform BNQ. A dynamic query unit allocation algorithm based on BNQRD can be obtained by extending the BNQ based algorithm that was just presented.

In BNQRD-based query unit allocation, each query unit is classified as being either I/O bound or CPU bound. This information is obtained from the input of the allocation algorithm (i.e., from the static planner). Since the query units to be allocated are classified into these two classes (I/O-bound and CPU-bound), all load-related metrics, including the initial load, the current load, the

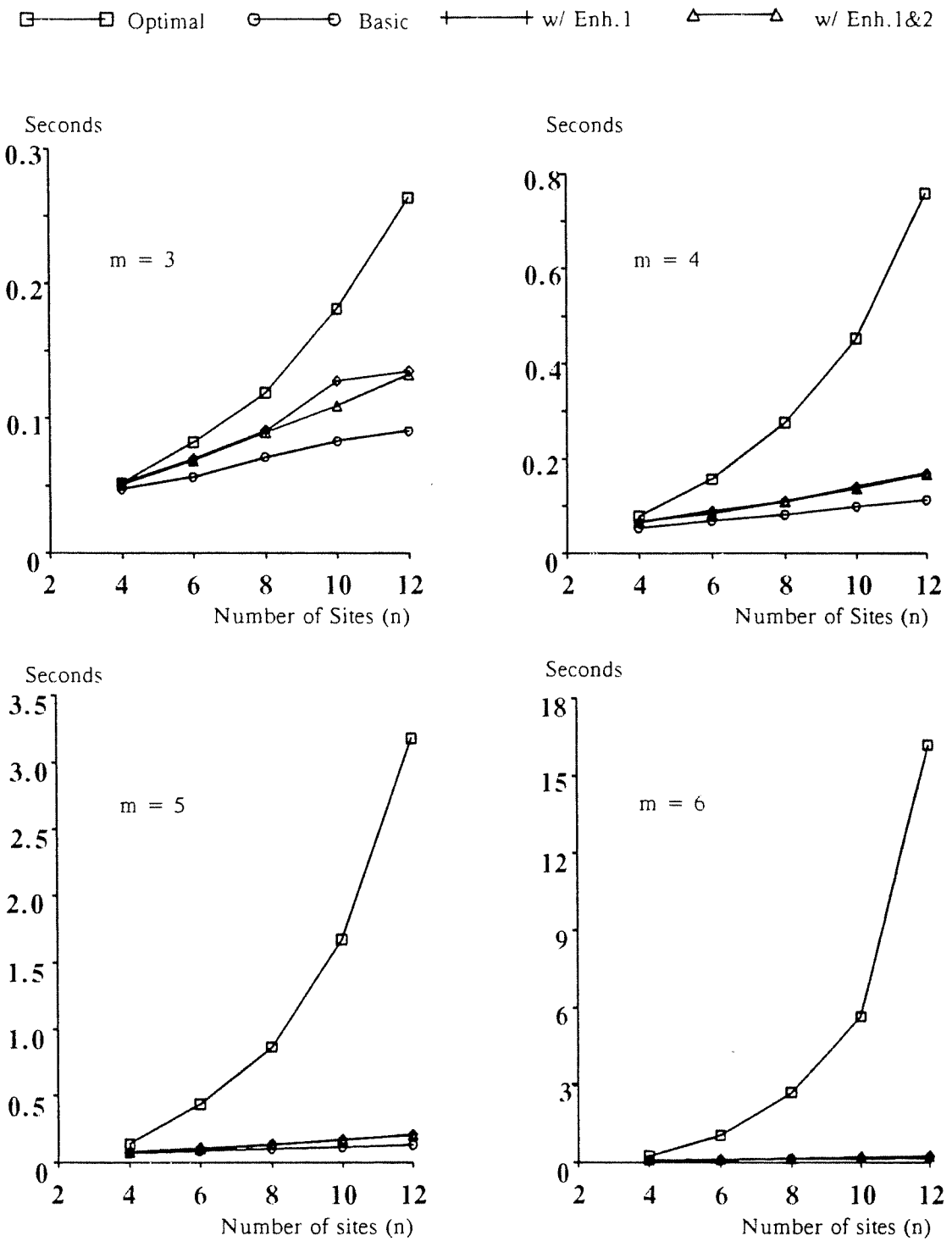

Figure 4.15: The elapsed time of the different algorithms.

load capacity, and the potential load, are extended to be two dimensional, with one dimension for I/O-bound query units and another for CPU-bound query units. The allocation procedure for a particular query unit is the same as in BNQ-based allocation, except that whenever the load-related parameters are referenced, those corresponding to the class of the query unit are used. Since the type of a query unit does not change the amount of data transferred between it and other query units, the communication-related parameters (such as benefit and potential benefit) are exactly the same as in the BNQ-based algorithm. The algorithm and its related procedures will therefore not be repeated here. Tests

Optimality of the BNQRD Heuristic Algorithm

m	n	total runs	Percentage of the Optimal Plans (%)		
			optimal plans	optimal on UBF	optimal on CC
3	4	100	98	99	99
	6		99	99	100
	8		97	98	99
	10		98	98	100
	12		95	96	99
4	4	100	97	99	98
	6		94	97	97
	8		97	98	99
	10		93	93	100
	12		91	94	97
5	4	100	96	100	96
	6		92	92	100
	8		92	95	97
	10		91	98	93
	12		88	95	93
6	4	100	98	100	98
	6		93	96	97
	8		93	95	98
	10		93	94	99
	12	87	90	92	98

Table 4.12: Optimality of BNQRD-based allocation.

similar to those described in Section 4.3.5 were conducted to investigate the optimality of the BNQRD algorithm. Table 4.12 presents the results of the tests. It can be seen that the BNQRD-based algorithm performs similar to the BNQ-based algorithm as far as optimality is concerned.

4.3.9. Summary of the Dynamic Allocation Phase

The dynamic query unit allocation problem was defined in this section. BNQ-based and BNQRD-based heuristic algorithms were developed to solve this problem. Tests indicate that the allocations chosen using the heuristic algorithms are quite good, and also that their execution time is much less than that of exhaustive search methods. After applying the allocation algorithm to the processing graph, each relation referenced in the graph is bound to a specific physical site. This site is recorded in a field of the relation node, and the processing graph is then passed to the third phase (the refining phase) which selects a join execution method for each of the distributed joins in the graph.

4.4. THE REFINING PHASE OF ALGORITHM LBQP

In the global processing plan, a join operation is specified as either a nested loops join or a sort-merge join. If the two relations of a join are allocated to two different sites in the second phase, a choice remains — both the nested loops join and the sort-merge join can be processed using either a semijoin-based execution method or a join-based execution method. The function of the refining phase of Algorithm LBQP is to make this decision for each of the distributed joins in the processing plan.

4.4.1. Semijoin and Join Methods

The experimental results described in Chapter 3 indicated that pipelined join and semijoin methods have similar performance. It is possible to execute the distributed joins in a processing plan using only pipelined join methods (i.e., the pipelined nested loops and sort-merge join algorithms). However, pipelined semijoin methods provide an opportunity to further reduce the communications costs in some cases.

Let $R_a \bowtie R_b$ be a join in a processing plan, wh $'_a$ and R_b are allocated to two different sites S_a and S_b. The major diffe es between the pipelined semijoin-based and join-based methods are as follows:

(1) The result sites in the two cases are different. The result relation ends up at site S_b for the pipelined join methods and at site S_a for the pipelined semijoin methods.

(2) The communications costs (the amount of data transferred between sites) are different. For the pipelined semijoin methods (PSNL and PSSM), the join column values of R_a are sent to S_b, and the matching tuples in R_b are sent back to S_a. For the pipelined join methods (PJNL and PJSM), all of relation R_a is sent to S_b.

The effects of these differences on a processing plan can be illustrated by an example. Consider a join $R_a \bowtie R_b$ in a database system with three sites, S_1, S_2, and S_3. Suppose that the query originates at S_1 and that the results are expected at the same site. Finally, assume that the allocation phase decides to access R_a at S_1 and R_b at S_2. If a pipelined join-based algorithm is used, the result relation of the join will end up at site S_2, and it will have to be transferred

back to site S_1. If the same distributed join is instead processed using a pipelined semijoin-based algorithm, the result relation will end up at site S_1, eliminating the extra cost for transferring the result relation to S_1. Similarly, if there is more than one join in a query, it is advantageous if the result site for one join can be the same as the processing site for the next join in the processing graph. This is the motivation for the refining phase of Algorithm LBQP.

One other issue need to be mentioned prior to describing the refining phase in detail. In the pipelined join-based algorithms, the task of merging tuples is performed at the inner site. In the pipelined semijoin-based algorithms, however, merging is done at the outer site. Thus, switching from a join-based algorithm to a semijoin-based one may cause inaccuracies in the CPU time estimates for the join processes (since it changes which one does the merging work). For the BNQRD-based query unit dynamic allocation algorithm, however, the important information about a query unit is the ratio of its I/O time and CPU time (which determines its class, I/O-bound or CPU-bound), and not its exact processing times. The cost of merging tuples in main memory is expected not to be so large that switching the merge site will change the class of a query unit.

4.4.2. The Refining Procedure

As described above, the refining phase explores the possibility of reducing the communications cost of a processing plan by using semijoin-based join methods. During the second phase of the algorithm (dynamic query unit allocation), the communications cost was taken to be 1 or 0 to distinguish remote processing from local processing when the BNQRD-based query unit allocation algorithm was used. In the refining phase, the communications costs for each

distributed join will be estimated using the information about its result size (from the processing graph) as follows.

Let R_a and R_b be two relations to be joined. Assume that R_a is the outer and R_b is the inner. The amount of data transferred by the pipelined join methods is:

$$data\_len\_join(R_a) = num\_tuples(R_a) \cdot tuple\_len(R_a) \quad (4.4.1)$$

The amount of data transferred by the pipelined semijoin methods is:

$$data\_len\_semijoin(R_a) = num\_tuples(Ra) \cdot attr\_len(R_a.A) \quad (4.4.2)$$

$$data\_len\_semijoin(R_b) = tuple\_len(R_b) \cdot num\_tuples(R_b \mid R_a) \quad (4.4.3)$$

Here, $num\_tuples(R_b \mid R_a)$ is the number of inner tuples in R_b which match the outer tuples from R_a. This number is determined not only by the semijoin selectivity of $R_a \rtimes R_b$, but also by the distribution of the join column values of the outer relation. In particular, duplication in the join column values of the outer relation will cause some of the inner tuples to be sent more than once.

Communications Cost		
Result Site	Semijoin (PSSM & PSNL)	Join (PJSM & PJNL)
$S_c = S_a$	data_len_semijoin(R_a) data_len_semijoin(R_b)	data_len_join(R_a) + result_size
$S_c = S_b$	data_len_semijoin(R_a) + data_len_semijoin(R_b) + result_size	data_len_join(R_a)
$S_c \neq S_a, S_b$	data_len_semijoin(R_a) + data_len_semijoin(R_b) + result_size	data_len_join(R_a) + result_size

Table 4.13: Total communications cost.

In addition to data transfers between the outer and inner relations, data transfers are also needed to send the join result to the next processing site (or the query's result site). If $R = R_a \bowtie R_b$ is the result relation of the join, and the join selectivity of $R_a \bowtie R_b$ is J_{ab}, the amount of result data to be transferred can be expressed as:

$$result\_size = J_{ab} \cdot num\_tuples(R_a) \cdot num\_tuples(R_b) \cdot tuple\_len(R) \quad (4.4.4)$$

If the join is part of a large query, the next processing site or the result site (if this join is the last one in the query), S_c, can be (1) site S_a; or (2) site S_b; or (3) some site other than S_a or S_b. Table 4.13 summarizes the communications costs for these three different cases in terms of equations (4.4.1) - (4.4.4).

Recall that the amount of data to be transferred is computed during the static planning phase and stored in the relation nodes and operation nodes of the processing graph. Operation nodes in the processing graph are modified (if necessary) by updating the *operation* field to be the selected method and the *result_site* field to be the resulting merge processing site. This refining process is shown in Figure 4.16. The input is the processing graph with the processing sites of the query units being selected in the dynamic allocation phase. For each join operation node, the functions *OpType(op_node)*, *OuterRel(op_node)* and *InnerRel(op_node)* return the type of the operation, and the outer and inner relations associated with the operation node in the processing graph; After refining, function *SetOp(op_node,X)* sets the *operation_type* field in the op_node to be operation X and function *SetResSite(op_node,S)* sets the *result_site* field in op_node to be S.

```
Algorithm Refining;
input : p_graph; (* the processing graph for the query with operation
                    nodes op_node's *)
output : p_graph; (* modified *)
  (* NextSite(p_graph, op_node) returns the next query unit's
       assigned processing site;
     TotalJoinCC(outer, inner, res_site) and
     TotalSemijoinCC(outer, inner, res_site) return the total
       communications costs computed as shown in Table 4.13;        *)
var  op : op_type;
     outer, inner, next_site : integer;
begin
  foreach op_node in p_graph do
    op := OpType(op_node);
    outer := OuterRel(op_node);
    inner := InnerRel(op_node);
    next_site := NextSite(p_graph, op_node);
    if ( op in {SM, NL} ) and (outer <> inner) then
       if TotalJoinCC(outer, inner, next_site) >
          TotalSemijoinCC(outer, inner, next_site) then begin
            SetResSite(op_node, outer);
            if (op = SM) then SetOp(op_node, PSSM)
            else SetOp(op_node, PSNL);
          end;
       else begin
            SetResSite(op_node, inner);
            if (op = SM) then SetOp(op_node, PJSM)
            else SetOp(op_node, PJNL);
          end;
  end;
end;
```

Figure 4.16: Algorithm Refining.

After the join methods have been determined for each join, there may be two or more consecutive semijoins in the processing graph. For these semijoins, processing can be simplified further. Suppose both joins in the query $(R_a[A=B]R_b)[D=C]R_c$ are to be processed using semijoin methods. The data transfer between the three relations is depicted in Figure 4.17 (a). The join column values $R_a.A$ are sent to R_b, and the matching tuples of R_b are sent back to R_a. The tuples from R_a and R_b are merged together at the site of R_a to form the result relation R_{ab}. The join column values of this intermediate result, $R_{ab}.D$, are then sent to R_c. The matching tuples of R_c are then sent back to R_a, where the final result is accumulated.

This 3-way join can be processed in another way, as shown in Figure 4.17 (b). The matching tuples of R_b can be sent directly to R_c instead of being sent

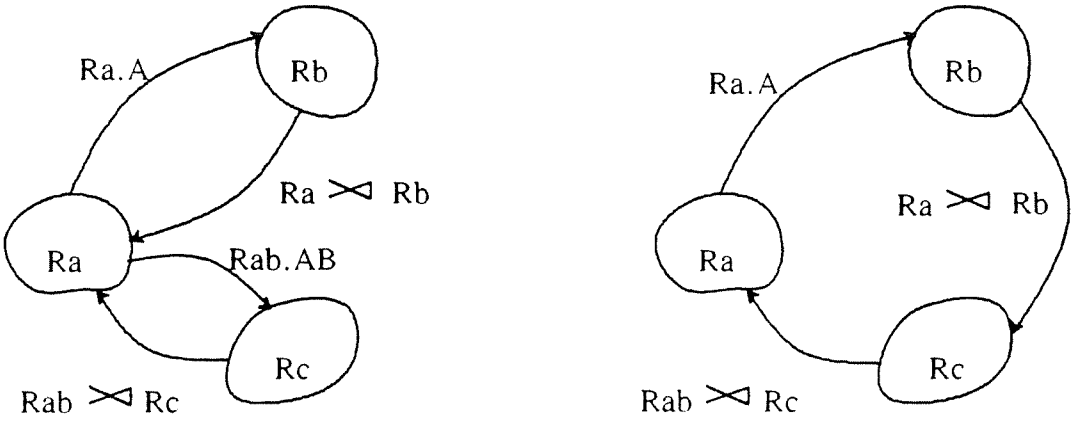

Figure 4.17: The processing of two consecutive semijoins.

back to R_a. The matching tuples in R_c are then selected and sent back to R_a along with the matching R_b tuples. (If the join column of the second join, $R_{ab}.D$, contains fields of R_a that are not in R_b, these fields must be sent to R_b along with $R_a.A$.) This strategy can also be used if more than two semijoins are adjacent in the processing graph. The refining phase could also be augmented to consider such possibilities.

The refining phase completes the global optimization phase for Algorithm LBQP. The resulting query processing plan defines the order of the joins, the join methods, the copies of relations to be referenced, and the data transfers needed between different sites. This plan is then distributed to all participating sites and the query is executed.

4.5. ALGORITHM LBQP: A SUMMARY

This chapter has described a new approach to query processing in locally distributed database systems, Algorithm LBQP. The approach consists of three phases, the static planning phase, the dynamic allocation phase and the refining phase. The salient features of this algorithm are as follows:

(1) Queries are initially optimized as though the system were a centralized database system. The main heuristics that led to this decision were (i) the optimal local join method is unaffected by the physical locations of the relations; (ii) the optimal choice for the outer and inner relations is unaffected by the physical locations of the relations; and (iii) the optimal join order for an n-way join is unaffected by the physical locations of the relations. Thus, the resulting logical plan can be converted into a good distributed plan by the dynamic allocation and refining phases.

(2) Load balancing is integrated with query processing. The notion of the load unbalance factor for a system was extended, and the query unit allocation problem was then defined based on this notion. The objective of this allocation was to minimize the load unbalance factor (the primary goal) as well as the overall communications cost (the secondary goal). Heuristic algorithms were then developed to solve the problem. Plans produced using these algorithms were compared with those produced by an exhaustive search method (which always found the optimal plan). The tests indicated that, in most cases, optimal allocation plans were found efficiently using the heuristic algorithms developed in this chapter.

(3) Since the choice between pipelined join methods and semijoin methods provides a possibility for further reducing the communications costs for distributed joins, both join-based and semijoin-based algorithms are considered for use in the final processing plan. The semijoin-based methods are used when they reduce the communications cost caused by transferring intermediate and final results.

The next chapter will examine the extent of performance improvements achievable using this proposed load-balanced query processing approach.

CHAPTER 5

QUERY PROCESSING WITH LOAD BALANCING: A SIMULATION STUDY

The best way to evaluate the load-balanced query processing algorithm proposed in the last chapter would be to implement it in a prototype of a locally distributed database system and then benchmark its performance. However, such a study would exceed the time and space limits of this thesis, and it would also require the availability of a multiuser prototype system. Instead, a simulation study was conducted to demonstrate the effectiveness of the dynamic allocation phase of the algorithm (which distinguishes the proposed algorithm from the previous research work in this area). The first part of this chapter given a simulation model for a locally distributed database system with partially replicated data. The details of the experiments and their results are then given in the rest of the chapter.

5.1. MODELING A DISTRIBUTED DATABASE SYSTEM

A simulation model for a distributed database system with fully replicated data was developed in Chapter 2 (shown in Figures 2.7 and 2.8). Since each relation is stored at every site in such a system, a query can be processed at any site without requesting data from other sites. In a system with partially replicated data, however, a query may reference relations that are not available at the local site and thus may need to access relations stored at other sites in order to complete its execution. Furthermore, when a query is processed in a pipelined fashion for improved performance, its query units may be allocated to different sites in the system for load balancing purposes; a query unit will then require

the intermediate results from its predecessor in the pipeline. The DB site model shown in Figure 2.8 of Chapter 2 thus needs to be modified to accommodate a more general model of distributed query processing.

5.1.1. The Generalized Model

Figure 5.1 depicts the generalized model used in this chapter. To model pipelined query execution, the model has to deal with the pipelined flow of data and the possibility that query units may have to block awaiting the arrival of data. Thus, in addition to the original queues for the CPU and I/O service centers and for message handling, a *blocked queue* and a *data pool* have been added to the model. The broken lines in the figure illustrate the flow of the intermediate results generated during query processing. The model of query generation and local query unit processing is the same as that described in Section 2.3 and will not be repeated here. However, the issues related to modeling more general queries and their pipelined execution do merit further discussion.

A user query is represented as a sequence of query units in the model, where each query unit accesses only one relation. The number of query units in a query is determined when it is initiated, and this can be varied in order to simulate different kinds of queries. For example, a selection or projection query consists of a single query unit, whereas a pipelined nested loops join operation (PJNL) can be modeled as a sequence of two query units. The first query unit in the join would access the outer relation and pass the results to the second query unit. The second query unit, after receiving the first page of data from the first query unit, begins its processing (i.e., fetching inner tuples using the incoming outer tuples).

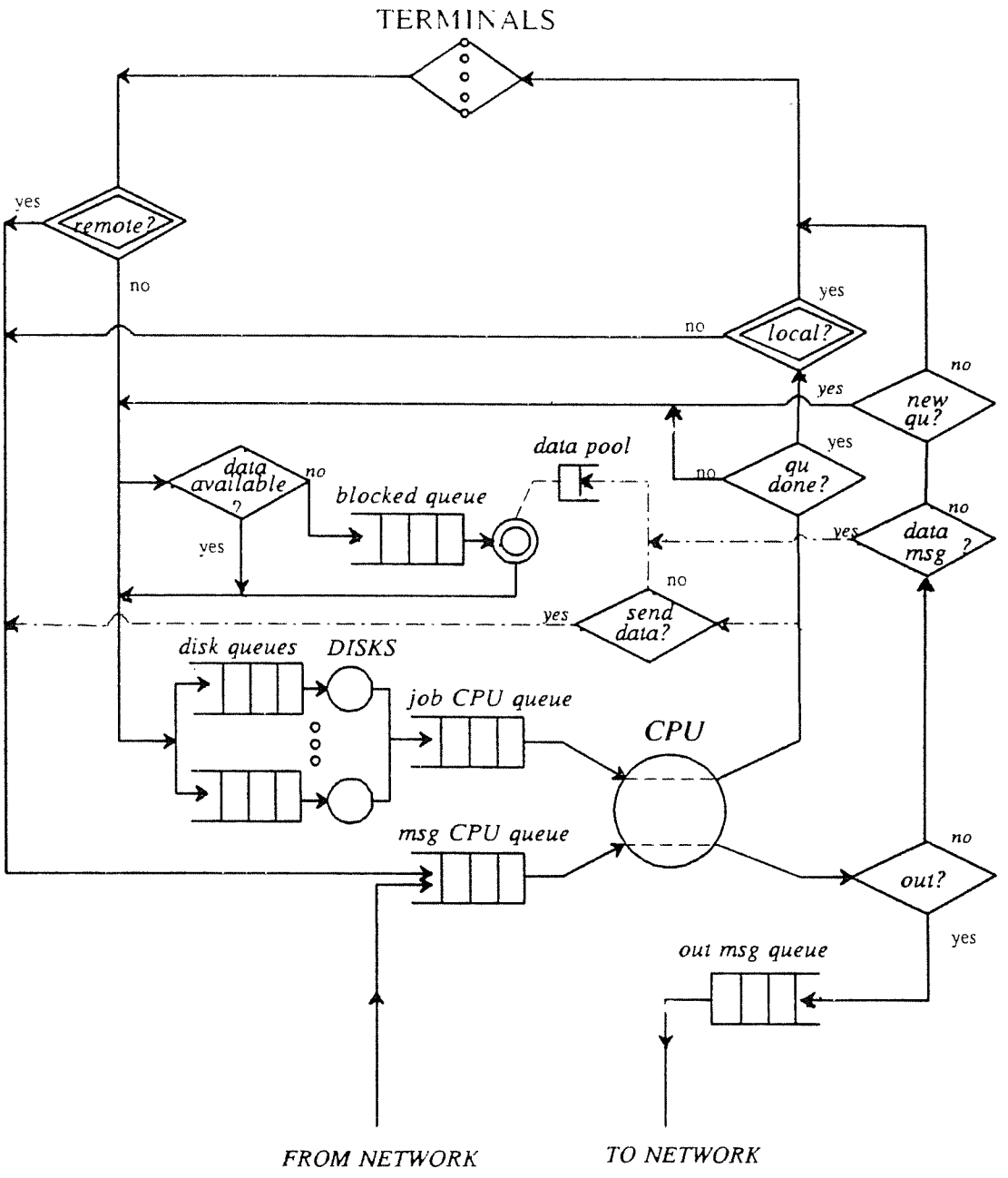

Figure 5.1: The generalized DB site model.

When a query is initiated by a terminal, the BNQRD-based query unit allocation algorithm is applied to decide the processing site for each query unit. 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 communications subnet. 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 the first data page is received 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. Each data page is identified by the query unit that produced it and the query unit that will receive it as input. 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 processing site of the next query unit, the data page will either be sent to a remote site via the communications subnet or placed in the data pool at its local site. Whenever a data page is added to the data pool, either by a message from another DB site or by a local query unit, the *blocked_queue* is checked. If the receiver of

the data page is already blocked, it leaves the blocked queue and proceeds to process the data.

Two types of messages are involved in this process. The first type of message is a *query message*, which carries information about query units to other sites for execution. A query message contains the description of a query unit, including its predecessor and successor in the pipeline, its execution time, its home site, etc. If a remotely executed query unit is the last one in the pipeline, a query message is also used to send a signal to its home site when it completes its execution. The second type of message is a *data message*. If two consecutive query units are processed at different sites, data messages are used to transfer the results from one site to the next. All messages are processed in the same manner as described in Chapter 2. Both incoming and outgoing messages first enter the *message CPU queue* to receive a certain amount of CPU service. 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 and disk service centers for execution or else back to the appropriate terminal if it is a query completion message. The communications subnet model is the same as that of Chapter 2.

From this description, it should be clear that this model is well-suited for simulating the pipelined processing of queries whose processing graphs are linear sequences of query units (such as the query shown in Figure 4.4 of Chapter 4). In particular, this model is a reasonably realistic representation of a sequence of PJNL joins. The simulation of more general queries would require a model with a more sophisticated flow control mechanism than that employed in

this model. However, this relatively simple model will be sufficient to demonstrate the effectiveness of the dynamic query unit allocation algorithm, which is the objective of this chapter.

5.1.2. Parameters of the Generalized Model

Table 5.1 lists the system parameters. In addition to the *num_sites* parameter of Chapter 2, the parameters *num_rels* and $storage\_sites_k$ are used to specify the number of relations in the system and the storage sites for each relation. In this study, it is assumed that all copies of a relation are identical.

The DB site parameters and the communications-related parameters are basically the same as those of Chapter 2. These parameters are given in Tables 5.2 and 5.3, respectively.

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

Table 5.1: System parameters.

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

Table 5.2: DB site parameters.

Communications-Related Parameters	
msg_setup	fixed amount of time needed to establish a connection
msg_cpu_rate	CPU time needed for transferring one byte of data
trans_rate	time needed for transferring one byte of data
query_descrip_size	size of a query message in bytes
data_page_size	size of a data message in bytes

Table 5.3: Communications-related parameters.

Query Parameters	
num_qtypes	number of different query types in the system
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 of query unit j being 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 unit
$page\_cpu\_time_{cpu}$	mean per-page CPU demand for CPU-bound query unit

Table 5.4: Workload parameters.

The workload of the distributed database system is described by three groups of parameters: query parameters, query unit parameters and class parameters. These parameters are listed in Table 5.4. The query parameters, including *num_qtypes*, num_qus_i and $qtype\_prob_i$, describe the general characteristics of the queries in the system. Queries are classified into *query types* based on their number of query units. The parameter *num_qtypes* is the number of dif-

ferent query types in the system. For each query type i, the parameter num_qus_i specifies the number of query units for this query type, and $qtype\_prob_i$ is the probability that a query submitted to the system 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 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. (This again indicates that the query processing model is most like the pipelined nested loops join method). The parameter $res\_fraction_{i,j}$ specifies the fraction of a result data page generated by each page processed. Finally, query units are still classified into two classes, I/O bound and CPU bound, according to their I/O and CPU service demands, as in Chapter 2. The parameter $class_{io}\_prob_{i,j}$ is the probability that query unit j will be I/O bound. For each class, the parameter *page_cpu_time* gives the mean CPU time (exponentially distributed) required to process a page of data.

5.2. SIMULATION DETAILS

The model described in the last section was implemented using the DENET simulation language [Livn85] and Modula-2. Since a large amount of CPU time was required for this study, the simulation was run concurrently using the remote UNIX facility on several of the DEC/VAX 11/750's of the Crystal multi-

computer [DeWi84b]. This section describes the details of the simulations, including the algorithms simulated, the parameter settings for the experiments, and the performance metrics used in this study.

5.2.1. Dynamic Query Unit Allocation Algorithms

The main purpose of this simulation study is to examine the performance of the dynamic query unit allocation algorithm LBQP of Chapter 4. For comparison purpose, three other algorithms, STATIC, RANDOM$_F$, and RANDOM$_P$ were also implemented. These algorithms select processing sites for query units as follows.

(1) LBQP. The simulation results of Chapter 2 indicate that information-based dynamic allocation performs better than simple BNQ-based algorithm. In this study, the BNQRD-based version of Algorithm LBQP from chapter 4 will be used to choose the processing site for each query unit in a query.

(2) STATIC. This algorithm tries to process a query unit at its local site if possible. If the relation referenced by a query unit is not available at the local site, a statically pre-determined site with a copy of the relation is selected as the processing site. In other words, in a system with partially replicated data, each relation is assumed to have a predetermined copy that will be used by query units whose local sites do not have a copy of the relation. (The term local site refers to either the home site of the query or the processing site of its predecessor in the pipeline.) This algorithm was motivated by existing distributed query processing algorithms that use a predetermined copy during query optimization [Bern81b].

(3) RANDOM$_F$. Algorithm RANDOM$_F$ is only applicable in systems with fully replicated data. It randomly selects a processing site for an entire query when the query is initiated, and all of its query units will be processed at that site.

(4) RANDOM$_P$. Algorithm RANDOM$_P$ is a random site selection scheme for systems where the data is partially replicated. This query unit allocation algorithm selects a processing site for a query unit as follows. (1) The first query unit is processed locally if the relation referenced by the query unit is locally available. (2) If the referenced relation is available at the site where the predecessor of a given query unit is processed, the query unit is processed at that same site to reduce the communications cost. (3) Otherwise, a processing site is randomly selected from among the sites where the relation is stored. RANDOM$_P$ selects copies in a manner similar to the copy identification algorithm proposed by Yu and Chang [YuCh83] — their algorithm statically selects the processing sites for a query, making the number of processing sites as small as possible to minimize the communications cost. RANDOM$_P$ is also similar to what system R* would do when all copies of each relation have identical acceess pathes. Finally, RANDOM$_P$ can also be thought as a variation of a simple *random splitting* load balancing algorithm, proposed by Wang and Morris [Wang85], which randomly selects a processing site for a task.

Table 5.5 gives an example that illustrates the differences between the three query unit allocation algorithms that are intended for partially replicated data. The system is assumed to consist of 3 sites, S_1, S_2, and S_3. Two relations, R_1 and R_2, are stored in the system as shown in Table 5.5(a). The predetermined

(a) System Parameters	
num_sites	3
num_rels	2
storage_sites	$R_1 : \{S_1, S_2\}$ $R_2 : \{S_2, S_3\}$

(b) Processing Sites				
Allocation Algorithm	Relation Referenced	Query's Originating Site		
		S_1	S_2	$S3$
STATIC	R_1	S_1	S_2	S_1
	R_2	S_2	S_2	$S3$
RANDOM$_P$	R_1	S_1	S_2	S_1 or S_2 (random)
	R_2	S_2 or $S3$ (random)	S_2	if R_1 is processed at S_2 then S_2 else $S3$
LBQP	R_1	S_1 or S_2 (based on load)		
	R_2	S_2 or $S3$ (based on load)		

Table 5.5: Different query unit allocation algorithms.

copies are R_1 at site S_1 and R_2 at site S_2. The processing sites selected by the algorithms for a join query $R_1 \bowtie R_2$ are listed in Table 5.5(b).

5.2.2. Parameter Settings

The model parameters used in the experiments of this chapter are listed in Table 5.6. 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.

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

Table 5.6: Parameter settings in the simulation.

In this study, the mean value of *page_cpu_time* is 4 times the *disk_time* for CPU bound query units and 1/4 of the *disk_time* for I/O bound query units. These values were suggested by experimental results for selection and join queries. For example, in experiments based on the Wisconsin Database [Bitt83], a selection that selected 100 tuples from a 10,000 tuple relation using a sequential scan was CPU bound; its CPU time was 79% of the total processing time. The same selection using a non-clustered index was I/O bound, and its disk time was 71% of the total processing time.

The settings for the remaining parameters are similar to those used in the simulation study described in Chapter 2. As in that chapter, the *think_time* parameter is varied to obtain different system loads in each experiment.

5.2.3. Performance Metrics

The mean waiting time of a query was the main performance metric of interest in the study of Chapter 2. This was defined as the difference between the mean response time and the service time of a query. In order to compare a pair of allocation algorithms L_1 and L_2, the mean waiting time improvement factor, $WIF(L_1, L_2)$, was introduced in Section 2.4.2.

In this study, a query can consist of more than one query unit, and these query units may be processed concurrently by different DB sites. Because of this parallelism, the mean response time of a query could actually be less than the sum of the service times for its query units. Furthermore, in addition to waiting for resources, a query unit may block during execution to wait for data to arrive from its predecessor. That is, the mean waiting time of a query, $\overline{W}$, can be thought of as consisting of two parts here:

$$\overline{W} = \overline{W}_R + \overline{W}_D$$

$\overline{W}_R$ is the time spent by the query units waiting for the desired resources, and $\overline{W}_D$ is the waiting time that would be caused by the absence of intermediate results if the query were executed alone in the system. From a load balancing viewpoint, $\overline{W}_R$ is obviously more interesting, as $\overline{W}_D$ is determined by the nature of the query and the choice of query allocation algorithm does not affect it.

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 first obtained by simulation. If all query units of a query are executed concurrently at different sites in "single-user mode", none of the query units will ever wait for resources, so the waiting time of the query will simply be $\overline{W}_D$. This response time, denoted as R_{su} (where "su" means in "single user mode"), will be the lowest mean response time which can be reached. The mean waiting time improvement factor in this simulation can thus be defined as:

$$WIF(L_A, L_B) = \frac{R_B - R_A}{R_B - R_{su}} \qquad (5.2.1)$$

R_A and R_B here are the query's response times when allocation algorithms L_A and L_B are used, respectively.

Throughput is another important performance metric, and dynamic query allocation algorithms may improve the throughput of a system as well. The notion of a throughput improvement factor, $TIF(L_A, L_B)$, can be similarly defined to measure this improvement:

$$TIF(L_A, L_B) = \frac{T_A - T_B}{T_B} \qquad (5.2.2)$$

T_A and T_B are the system's throughput when allocation algorithm L_A and L_B are used, respectively.

For the experiments presented here, both the throughput of the system and the response times of completed queries were measured. The response time R_{su} was obtained by running separate simulations. $WIF(L,STATIC)$ and $TIF(L,STATIC)$ will be computed from these results.

5.3. 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. In Experiment 1, a single fixed query type is used to investigate the performance of the algorithms in systems with different numbers of copies of the relations. In Experiment 2, the workload is a mixture of queries comprised of different numbers of query units. Lastly, Experiment 3 attempts to illustrate the effectiveness of dynamic load balancing in a system with different (static) loads at the various DB sites.

5.3.1. Experiment 1: Varying Data Replication

The query used in this experiment consists of three query units. The first query unit is CPU bound and the other two are I/O bound. The query corresponds to a 3-way join $(Ra \bowtie R_b) \bowtie R_c$: Query unit 1 is a sequential scan of the outer relation R_a (CPU bound). The result is pipelined to query unit 2, which uses the incoming tuples to fetch inner tuples from R_b through a non-clustered index (I/O bound). The resulting tuples are sent to query unit 3 which performs the join with R_c (again I/O bound). The service demands of the three query units are listed in Table 5.7. The other parameters used are

those listed in Table 5.6.

The relations in the system and their storage sites are listed in Table 5.8. Four tests were performed in Experiment 1, each with a different level of data replication. In Test 1, all three relations were replicated at every site (Case 1 in Table 5.8). Test 2 had 4 copies of each relation, and Test 3 had 2 copies of each relation (Case 2 and 3 in Table 5.8). In Test 4, the most heavily used relation (R_1) was fully replicated in the system and the other two relations each had only two copies (Case 4 in Table 5.8).

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 5.7: The test query for Experiment 1.

Relations and Their Storage Sites				
Site	Case 1	Case 2	Case 3	Case 4
S_1	$\{R_1, R_2, R_3\}$	$\{R_1^*, R_3\}$	$\{R_1^*\}$	$\{R_1\}$
S_2	$\{R_1, R_2, R_3\}$	$\{R1, R_3\}$	$\{R_1\}$	$\{R_1\}$
$S3$	$\{R_1, R_2, R_3\}$	$\{R_1, R_2^*\}$	$\{R_2^*\}$	$\{R_1, R_2^*\}$
$S4$	$\{R_1, R_2, R_3\}$	$\{R_1, R_2\}$	$\{R_2\}$	$\{R_1, R_2\}$
$S5$	$\{R_1, R_2, R_3\}$	$\{R_2, R_3^*\}$	$\{R_3^*\}$	$\{R_1, R_3^*\}$
$S5$	$\{R_1, R_2, R_3\}$	$\{R_2, R_3\}$	$\{R_3\}$	$\{R_1, R_3\}$
(Relations with *'s are the preassigned copies)				

Table 5.8: Relations and their storage sites.

Figure 5.2 presents the results of Test 1, where all three relations are fully replicated. The mean response time (Figure 5.2 (a)), the waiting time improvement factor $WIF(LBQP, STATIC)$ (Figure 5.2 (b)), the throughput of the system (Figure 5.2 (c)), and the throughput improvement factor $TIF(LBQP, STATIC)$ are given as functions of think time. Since the data is fully replicated and the workloads are the same at every site, all sites in the system have the same CPU and disk utilizations. The CPU utilization is given in Table 5.9 for reference.

The first observation from Figure 5.2 is that the load-balanced query unit allocation algorithm (LBQP) leads to a significant decrease in waiting time with respect to the STATIC algorithm. The response time of the query approaches R_{su} as the system load decreases. Greater *WIF* values are obtained when the system is less heavily loaded. When the *think_time* is greater than 16 seconds (corresponding to a CPU utilization of less than 0.5 at every site), the waiting time improvement factor is greater than 50%, but as the system becomes more heavily loaded, the improvement becomes smaller. With 4 seconds of *think_time* (where the CPU utilization is about 0.92), LBQP actually performs slightly worse than STATIC. This is explainable since the result fraction is relatively large (0.2) in the current parameter settings. A large intermediate result (an average of 4 pages per query unit) is generated and transferred if the query units are dynamically allocated. Another observation from this test is that, in a system with fully replicated data and a uniform workload, randomly selecting processing sites is unlikely to be profitable. The response time for $RANDOM_F$ is always larger than that of STATIC. ($WIF(RANDOM_F, STATIC)$ is less than 0 and is not shown in the figure). This is because a randomly selected remote site is likely to be as heavily loaded as the local site, and transferring the query units

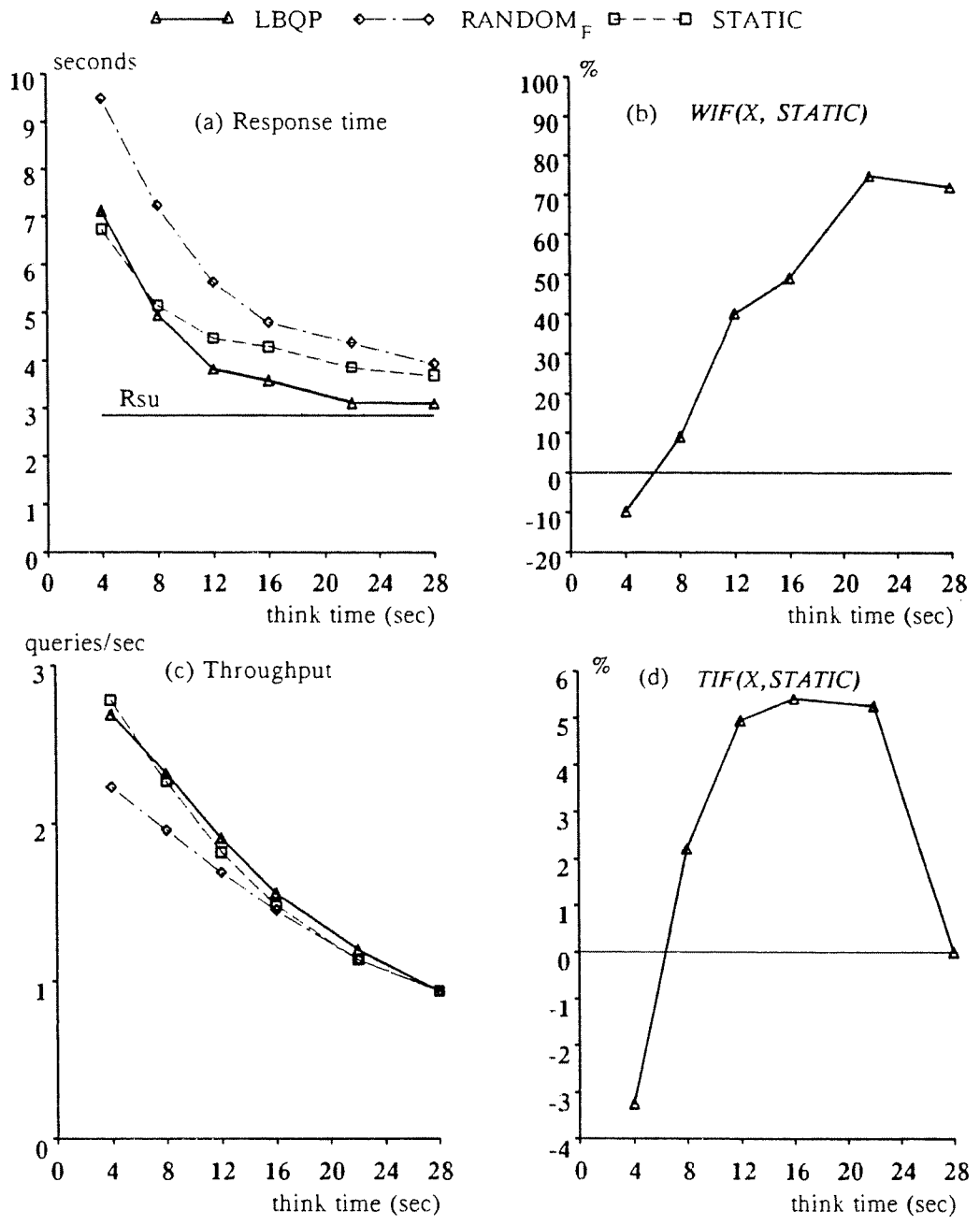

Figure 5.2: Results of Experiment 1, Test 1.

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 5.9: CPU utilization in Experiment 1, Test 1.

and result data only increases the communications cost.

As for the system throughput, $RANDOM_F$ is again worse than STATIC. With algorithm LBQP, the throughput of the system is increased by a few percent, with $TIF(LBQP, STATIC)$ increasing a little as the system load decreases. It reaches a maximum and then decreases along with the decrease in the system load. This can be explained as follows: If the system is lightly loaded, its throughput is limited by the think time (i.e., by the query arrival rate). No matter how fast queries can be processed using LBQP allocation, no significant improvement in throughput is possible. When the system is heavily loaded, the throughput is determined by the queuing time of the queries. At this end of the spectrum, however, LBQP does not effectively decrease the response time, so the throughput improvement factor is again small. The maximum improvement in throughput is therefore obtained under a medium load, as shown in Figure 5.2 (d).

Tests 2 and 3 of Experiment 1 studied the performance of the query unit allocation algorithms in cases where the relations are partially replicated. Each relation has 4 copies in test 2, and in test 3 the number of copies decreases to 2. Figures 5.3 and 5.4 show the results of these two tests.

Several observations can be made from these results. First, the response time for the STATIC algorithm increases rapidly when the number of copies is

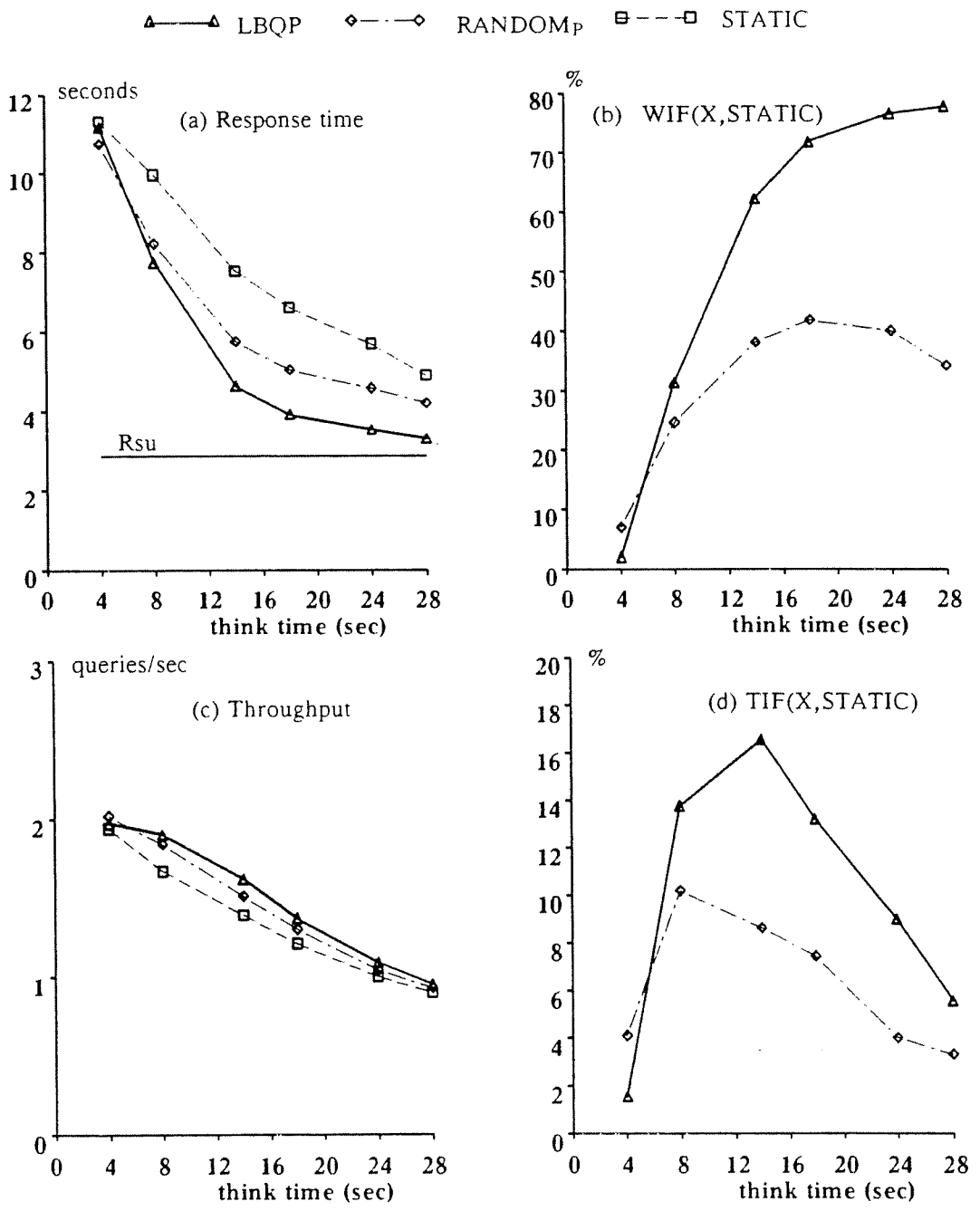

Figure 5.3: Results of Experiment 1, Test 2.

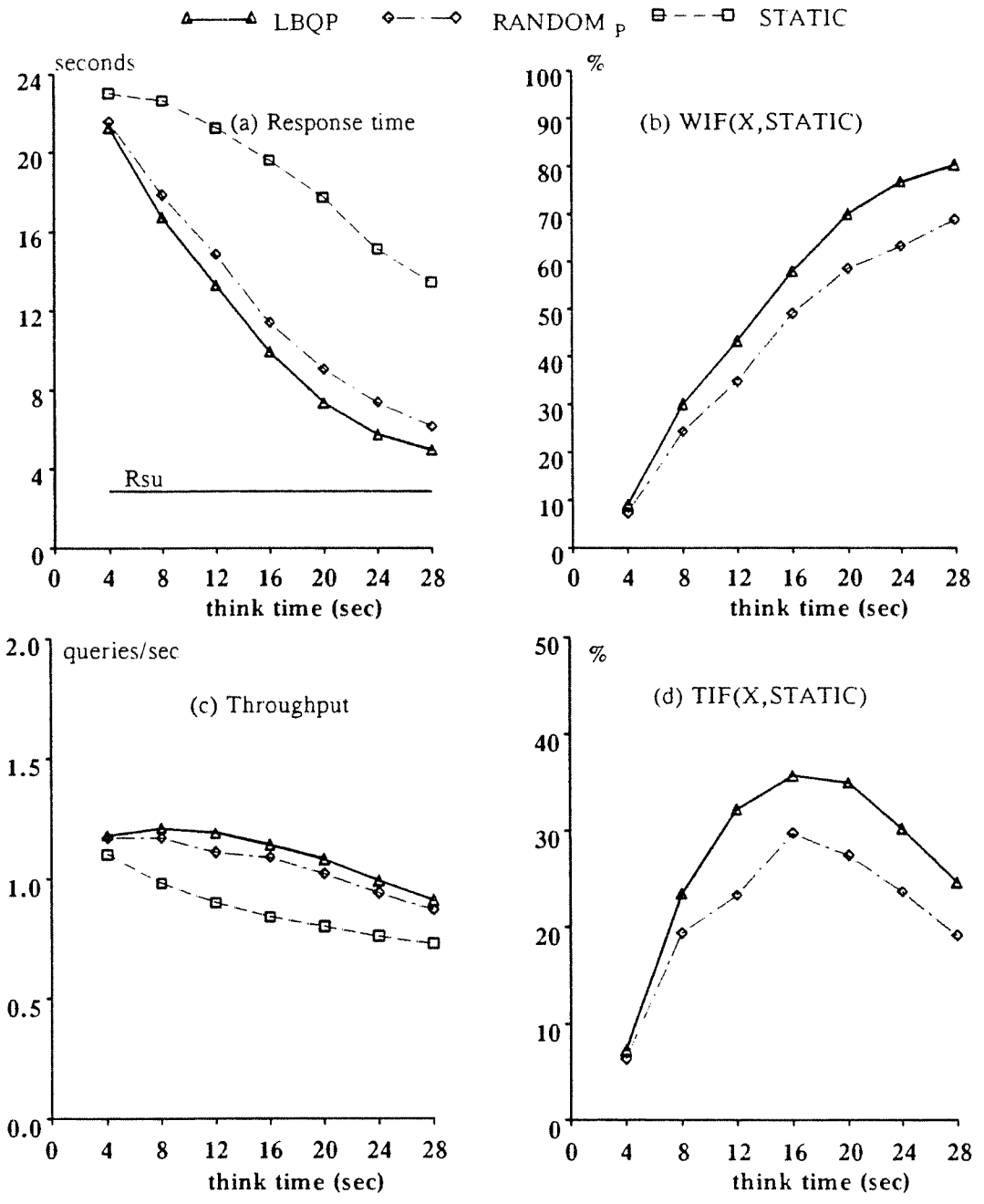

Figure 5.4: Results of Experiment 1, Test 3.

decreased, especially with a short *think_time*. This is caused by competition for accessing the preassigned copies. It is exactly this resource contention problem that the load-balanced query processing algorithm is designed to alleviate. It is evident that LBQP causes the response time to decrease significantly in both tests, and also that the waiting time improvement factor increases with a decrease in the system load. In Test 2, when the average CPU utilization of the four most heavily loaded sites is 0.85 (at a think time of 8 seconds), $WIF(LBQP, STATIC)$ is greater than 30%. When the think time is increased to 14 seconds (where the average CPU utilization is 0.68), the value of $WIF(LBQP, STATIC)$ increases to 60%. The improvement in Test 3 is similar. The rate of this increase is a little higher with 4 copies per relation, as more copies provide more opportunities to allocate a query unit to a lightly loaded site. However, in both cases the lowest response time reached by LBQP is still higher than that of the fully replicated case (Test 1).

Another observation is that $RANDOM_P$ performs better than STATIC in Tests 2 and 3 because the partially replicated data makes the system unevenly loaded; randomly picking processing sites helps to equalize the system load. In the case where each relation has just two copies (Test 3), the performance of $RANDOM_P$ is close to that of LBQP. This is because $RANDOM_P$ has a 50 percent chance (statistically) of making the right selection from the perspective of load balancing in this case.

In both tests, a greater improvement in system throughput is observed than that of Test 1. In Test 2, the maximum $TIF(LBQP, STATIC)$ is 16.55%, and in Test 3, it is as high as 35%. This clearly shows that the dynamic query unit allocation algorithm effectively eliminates resource contention at heavily loaded

sites, so both the response time for queries and the throughput of the system are significantly improved.

It can be seen from the parameter settings that the first query unit is likely to be the bottleneck in the execution of the overall query in these tests — the service time of the first query unit is two thirds of the total processing time for the query. In Test 4 of Experiment 1, only relation R_1 (which is referenced by the first query unit) was replicated at every site, and the other relations each had just two copies (Case 4 in Table 5.8). Figure 5.5 presents the results of this test. Figure 5.5 (b) shows that LBQP provides a significant improvement in waiting time, and Figure 5.5 (a) indicates that fully replicating R_1 improves the response time dramatically for both the STATIC and LBQP algorithms. Figure 5.6 shows the response times obtained in Tests 1 and 4 together, clearly showing this point. The response time of the query in the partially replicated case is close to that of the fully replicated case when the LBQP algorithm is used to allocate the query units.

5.3.2. Experiment 2: Mix of Query Types

In Experiment 1, the workload used in the tests consisted of a single query type. Experiment 2 investigates the performance of LBQP for more general workloads. The workload used in this experiment is a mix of queries with different numbers of query units. The parameter settings for Experiment 2 are shown in Table 5.10. A newly generated query will consist of 1, 2 or 3 query units (with probabilities 0.5, 0.3 and 0.2, respectively). For each query unit, the probability of it being I/O bound is 0.5, and otherwise it will be CPU bound. The service demands for I/O bound and CPU bound query units are the same as

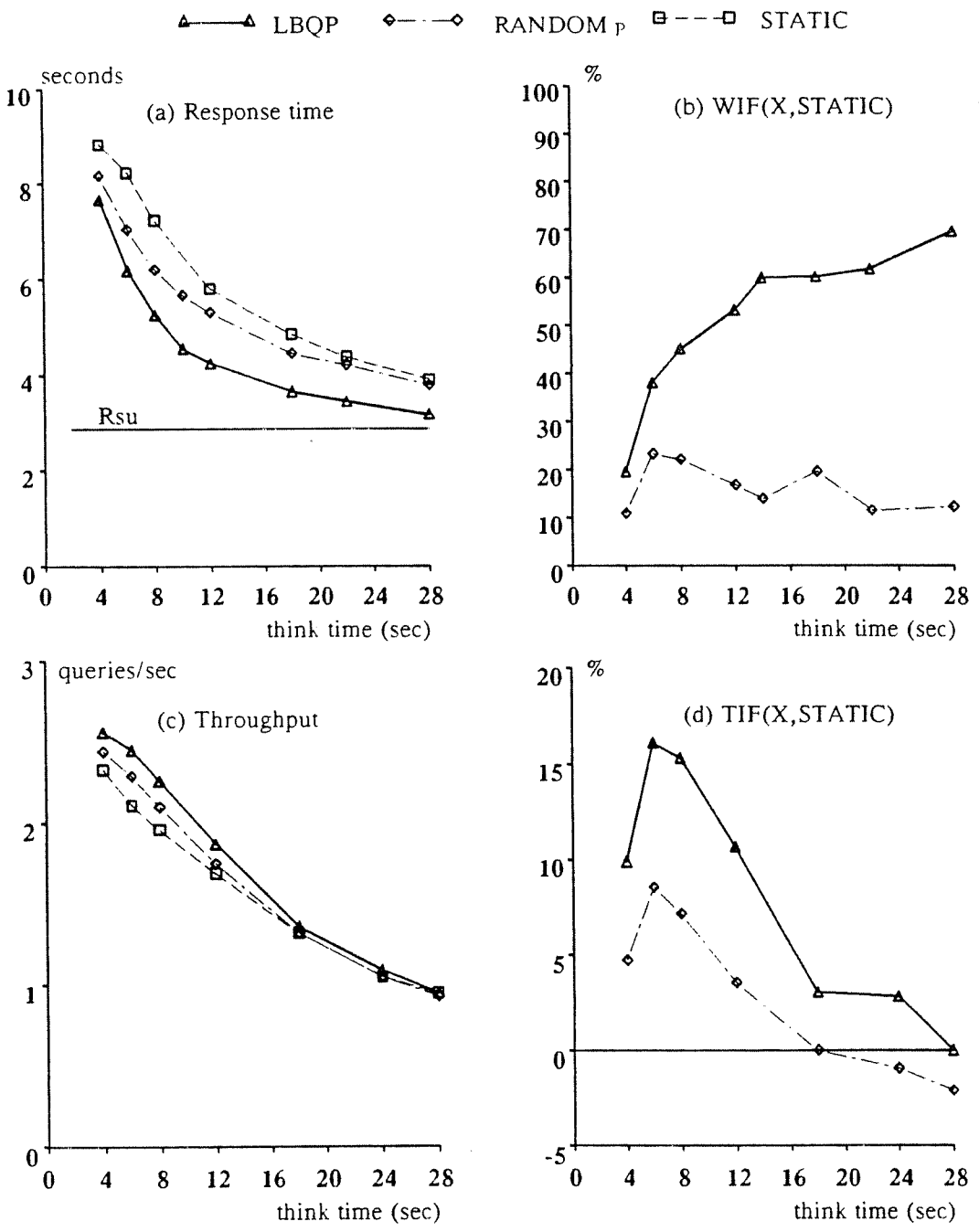

Figure 5.5: Results of Experiment 1, Test 4.

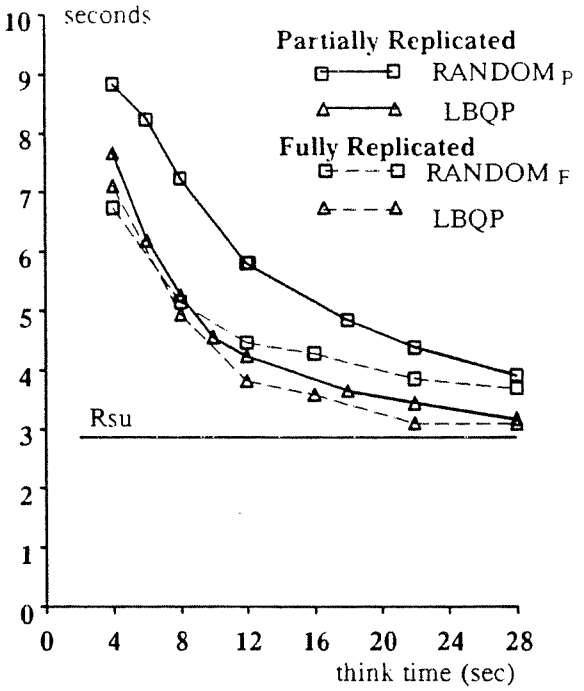

Figure 5.6: Comparison between Tests 1 and 4 response times.

Query Parameters			
num_qtypes	3		
Query Type	1	2	3
num_qus	1 (R_1)	2 ($R_1 \bowtie R_2$)	3 ($R_1 \bowtie R_2 \bowtie R_3$)
qtype_prob	0.5	0.3	0.2
Query Unit Parameters			
Query Unit	1	2	3
num_reads	20 pages	5 pages	5 pages
res_fraction	0.2	0.2	0.2
$class_{io}$*_prob*	0.5	0.5	0.5
Class Parameters			
Class	I/O Bound	CPU Bound	
page_cpu_time	5 msec	80 msec	

Table 5.10: The workload for Experiment 2.

in Experiment 1.

Two tests were performed in this experiment, one with fully replicated data (Case 1 in Table 5.8) and the other with partially replicated data with 4 copies of each relation (Case 2 in Table 5.8). Figures 5.7 and 5.8 present the results obtained in these two tests. Again, the dynamic query unit allocation algorithm (LBQP) provides a significant improvement in response time. Similarly, the throughput improvement factors in these two tests follow the same trends as in Experiment 1. In the fully replicated case, the throughput improvement factor is small. In the partially replicated case, however, throughput as well as response time is improved by dynamic query unit allocation. When the *think_time* is 6 seconds, LBQP increases the throughput by 22% and decreases the waiting time by 65%. When the *think_time* is increased to 14 seconds, the decrease in waiting time is 75% with a 7% throughput increase.

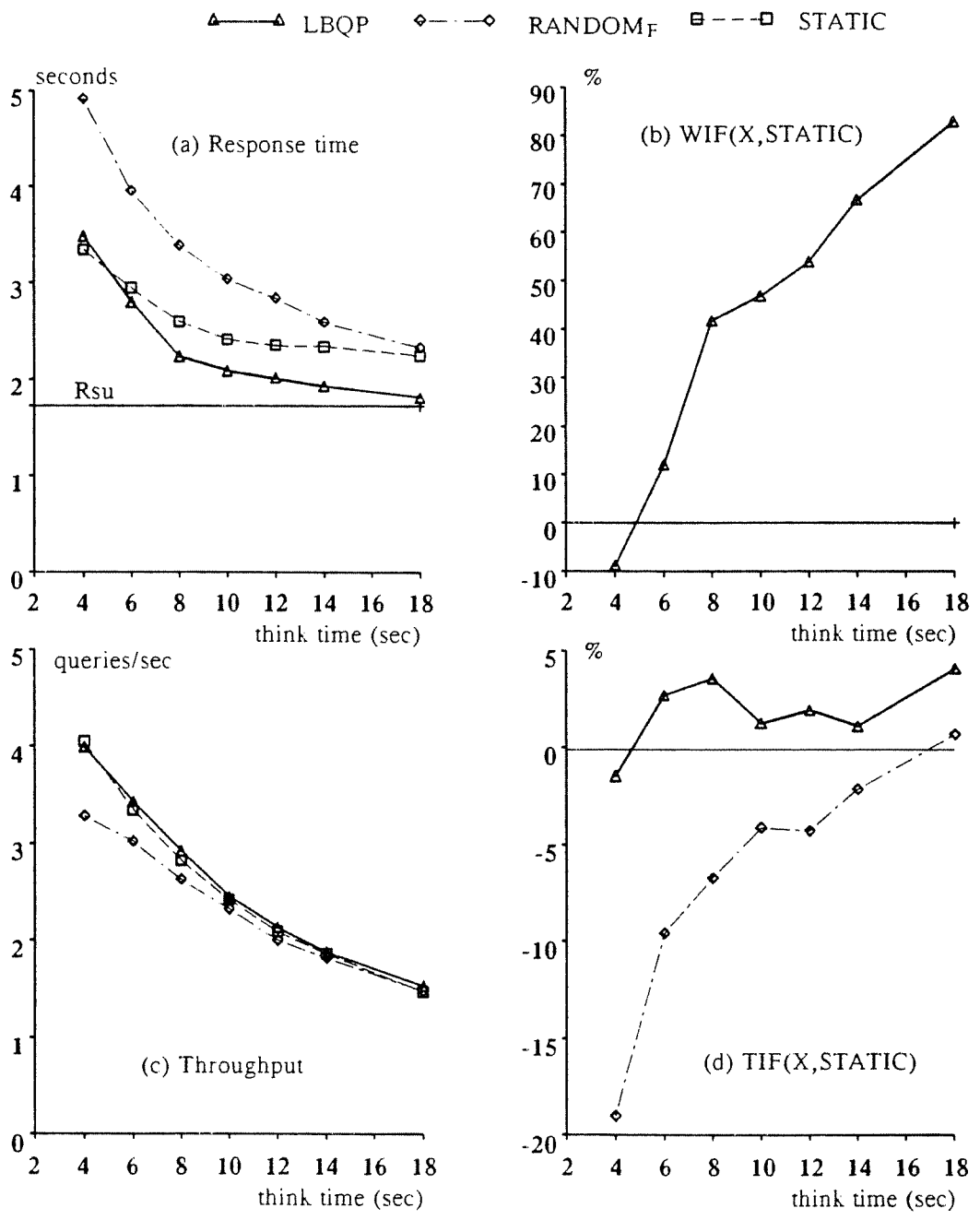

Figure 5.7: Results of Experiment 2, Test 1.

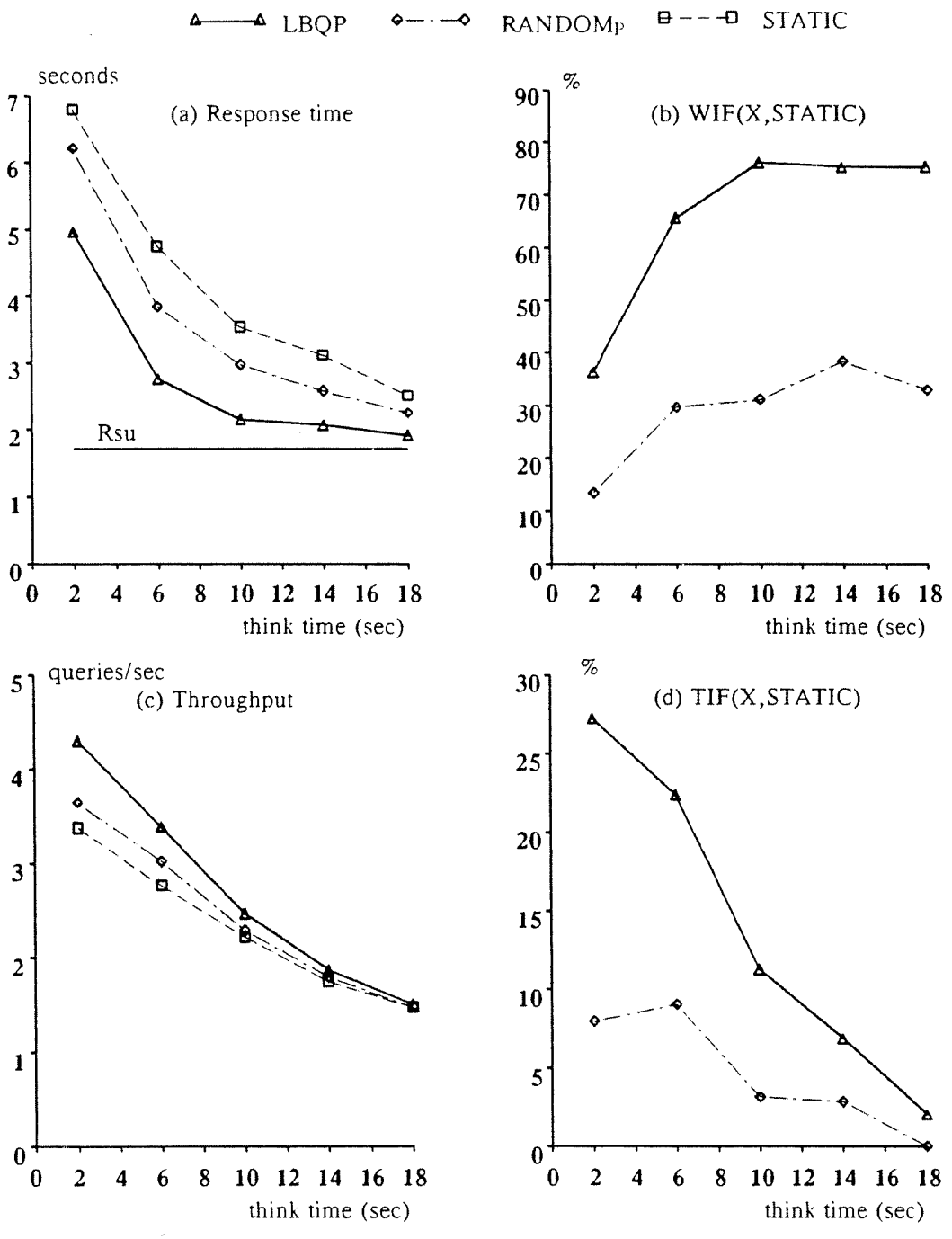

Figure 5.8: Results of Experiment 2, Test 2.

The results of Test 2 can be better understood by looking at the system's utilization. Figures 5.9 (a) and (b) show the CPU utilization for the 6 sites in the system for *think_time* values of 6 and 14 seconds. If the STATIC query allocation algorithm is used, the two sites that have the preassigned copies of relations R_1 and R_2 are the most heavily loaded sites[†]. With a short *think_time*, queries are queued at these sites leading to long response times and low throughput. Dynamic query allocation not only helps to equalize the CPU load among the sites in the system, but it also increases the system-wide CPU utilization: In this case, the average CPU utilization over the 6 sites used for processing queries (excluding the CPU time consumed by messages) increases from 0.558 to 0.695. This is about a 25% increase over the STATIC allocation algorithm. Figure 5.9 (b) also shows equalization of the CPU utilizations of the sites, but the system-wide average utilization only increases from 0.383 to 0.392, or about 2%, in the long *think_time* case.

5.3.3. Experiment 3: Non-Uniform Query Arrival Rates

In Experiments 1 and 2, every site in the system had the same (statistical) workload. Experiment 3 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) is set to 1 at three of the sites (sites S_1, $S3$, and $S5$) in order to obtain non-uniform arrival rates for queries. The relations in this experiment are fully replicated (Case 1 in Table 5.8). The results of Experiment 3 are illustrated in Figure 5.10, including the response time,

[†]Most queries have only one or two query units, and therefore access only R_1, and possible R_2.

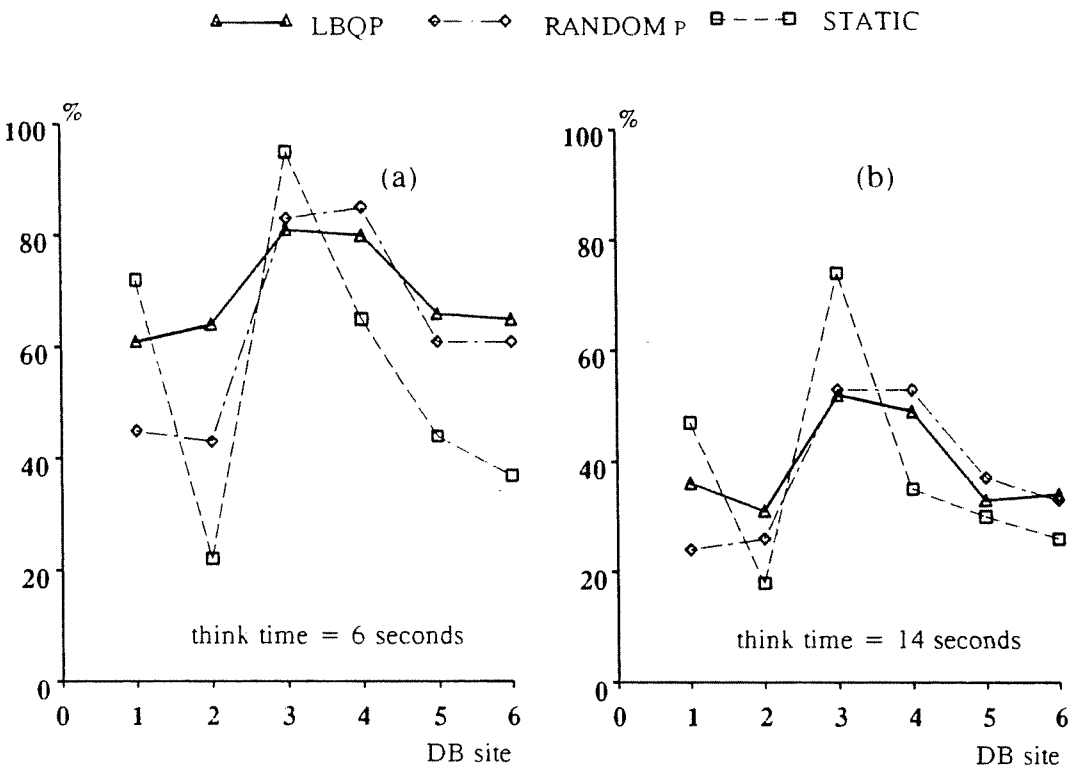

Figure 5.9: CPU utilization (Experiment 2, Test 2).

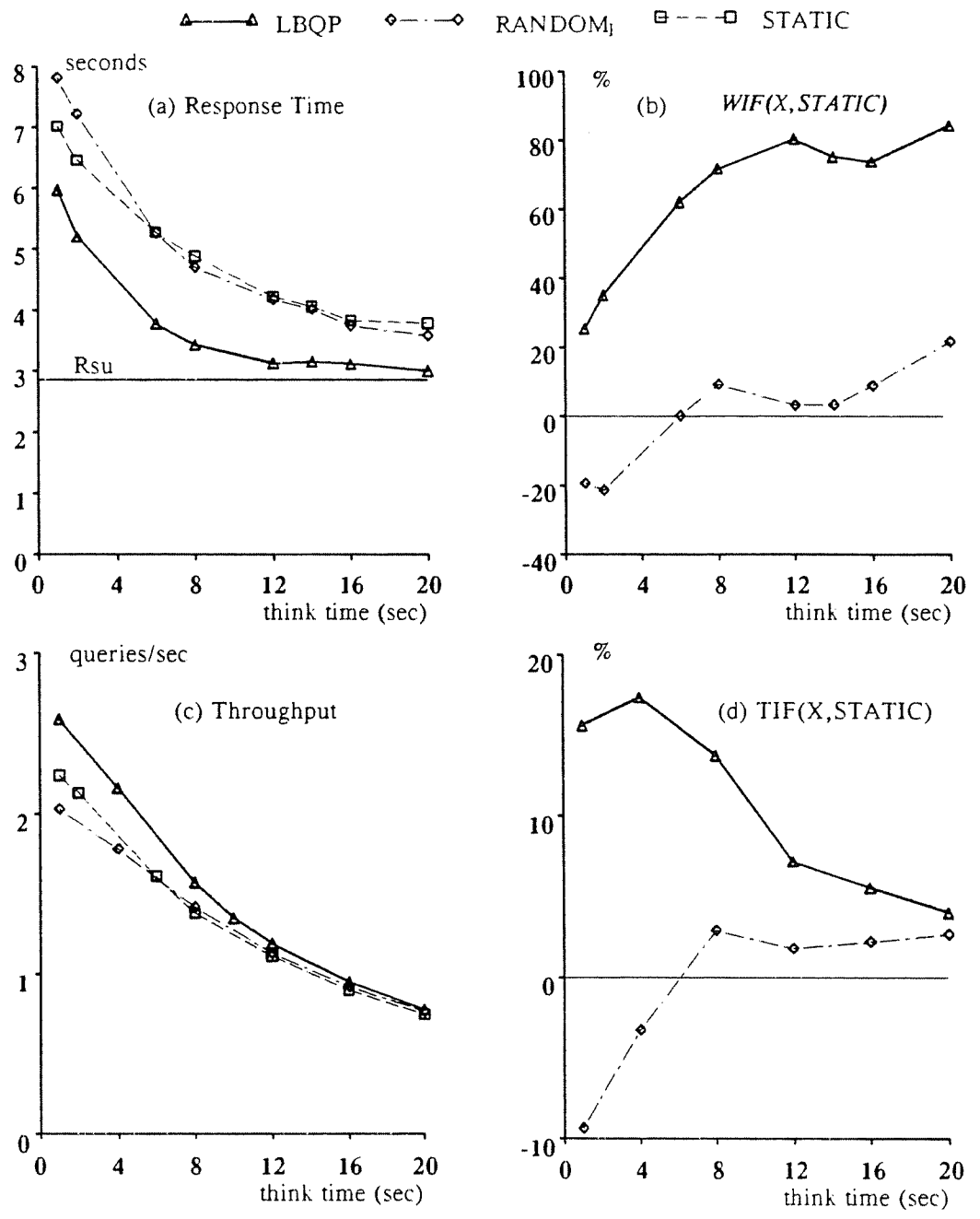

Figure 5.10: Results of Experiment 3.

throughput, and their improvements due to dynamic query unit allocation. These figures show the same trends as in previous tests with partially replicated data. 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 (where there was fully replicated data and a uniform workload), with slight improvements in its *WIF* and *TIF* being observed when the system is lightly loaded. LBQP performs much better, as shown by both the *WIF*(*LBQP*, *STATIC*) and *TIF*(*LBQP*, *STATIC*) metrics. With a *think_time* of 1 second (where the system-wide average CPU utilization is about 0.86), *WIF*(*LBQP*, *STATIC*) exceeds 25%, and *TIF*(*LBQP*, *STATIC*) is greater than 15%. When the system load increases, the waiting time improvement increases rapidly and the throughput improvement decreases somewhat. When the *think_time* is 8 seconds, the value of *TIF* is still a little over 13%, while the *WIF* exceeds 70%. It is clear that algorithm LBQP improves the performance of the system significantly here.

5.4. SUMMARY

This chapter described a simulation model for a distributed database system with partial data replication. In this model, a query was represented as a sequence of query units that executed concurrently in a pipelined fashion. The parameters used to represent the workload and the performance metrics used to study the performance of the load-balanced query allocation algorithm (LBQP) were discussed.

Three simulation experiments were performed to investigate the performance of Algorithm LBQP. Three other algorithms, STATIC, RANDOM$_P$,

and RANDOM$_F$, each of which can be viewed as the site selection method used by one or more existing query allocation algorithms, were studied as reference points. A number of conclusions were drawn from the results of the experiments. First, Algorithm LBQP indeed improves performance, decreasing the response time for queries and increasing the throughput of the system. This performance improvement was observed in a variety of cases with different levels of data replication, different query mixes, and non-uniform query arrival rates. (The throughput improvement in cases with fully replicated data and a uniform workload was not as large as in other cases). Second, both partial data replication and non-uniform workloads cause queries to queue up at certain sites, causing the performance of the system to degrade. The dynamic query unit allocation algorithm performed very well in these cases, as indicated by both its waiting time improvement factor and its throughput improvement factor. Finally, the RANDOM$_F$ algorithm did not perform as well as STATIC in cases with fully replicated data. With partially replicated data, however, the RANDOM$_P$ algorithm (which randomly selects processing sites) performed better than the STATIC algorithm but worse than LBQP.

CHAPTER 6

CONCLUSIONS

6.1. SUMMARY OF RESULTS

Most all of the distributed query processing techniques proposed in the literature have been based on the static characteristics of a distributed database system. "Optimal" processing plans are obtained without considering the dynamic characteristics of the system such as its load. This thesis proposed a new approach to distributed query processing in locally distributed database systems: load-balanced query processing (LBQP). This approach is novel in that it integrates distributed query processing with load balancing. The processing plan for a query is first optimized to minimize the total processing time. Processing sites are then selected in such a way that the system load is balanced, leading to better performance as compared with previous static distributed query processing methods.

Chapter 2 of this thesis presented a simulation model for distributed database systems with fully replicated data, and it also presented three basic dynamic query allocation algorithms: BNQ ("Balance the Number of Queries"), BNQRD ("Balance the Number of Queries by Resource Demands") and LERT ("Least Estimated Response Time"). Each of these algorithms was studied using the simulation model, and the results that were obtained clearly showed the potential of dynamic query allocation. Even in systems with homogeneous sites and workloads, it was found that a significant decrease in the mean waiting time for queries can be obtained when queries are allocated to sites based on load

information. Among the three dynamic query allocation algorithms, the results indicated that algorithms which use information about query resource demands (e.g., BNQRD and LERT) lead to better performance than a simple load balancing approach that does not use such information (e.g., BNQ). Lastly, in addition to decreasing the mean waiting time for queries, it was found that dynamic allocation also makes the system treat queries with differing resource demands more fairly.

Chapter 3 presented an empirical performance study of distributed join methods. In this study, a testbed was built using a locally distributed computer system, the Crystal multicomputer. Eight distributed join methods, which were combinations of sequential or pipelined join methods, methods using the semijoin or the full join operator, and the nested loops join algorithm or the sort-merge join algorithm, were implemented on the testbed. The results obtained suggest that pipelined query processing methods, based either on the full join operator or on the semijoin operator, are the best methods for processing distributed joins. Nested loops join methods (using a join column index) performed better than their sort-merge counterparts in most of the tests. Sort-merge join methods won only when the relations to be joined were very large, when the savings due to the merge scan compensated for the sorting costs. The results also indicated that communications cost is not a dominant factor with respect to the performance of join algorithms in a local network.

Based on these results and on other related research work, Chapter 4 presented a new approach to query processing, the load-balanced query processing algorithm LBQP. This algorithm generates a query processing plan in three phases. The first phase is the static planning phase. A logical plan that

minimizes total processing cost in the static sense is generated in this phase. In the second phase, the dynamic allocation phase, a dynamic query unit query allocation algorithm is applied to the logical plan to select the processing site (i.e., physical copy) for each relation referenced by the query. Finally, the refining phase chooses between join and semijoin methods for the distributed joins in the plan. The distinguishing features of this approach are:

(1) A query in a locally distributed database system is statically optimized in the same way as in a centralized database system. The storage sites of the relations referenced by the query are ignored during optimization, and a good distributed plan is still obtained in the end.

(2) The load unbalance factor is used as a quantitative measure of the "unbalancedness" of a distributed database system. Heuristic methods are used to allocate a sequence of query units to sites in such a way that the unbalance factor of the system is minimized. Communications costs are also minimized to the extent possible without adversely affecting the unbalance factor. It was shown that the proposed algorithms usually generate the optimal allocation plan, and that they do so within a reasonable amount of time.

(3) Both the semijoin operator and the full join operator are used in processing plans. This provides more opportunities to reduce the communications cost while selecting the right processing site for load balancing purposes.

The dynamic query unit allocation phase is the essence of the load-balanced query processing approach. In order to demonstrate the effectiveness of the proposed dynamic query unit allocation algorithm, Chapter 5 presented a generalized simulation model for queries whose processing plans consist of linear sequences of query units to be processed in a pipelined fashion. Simulation

results indicate that the dynamic query unit allocation algorithm can both increase the system throughput and decrease the response time for queries in systems with either fully or partially replicated data. The decrease in response times, measured using the mean waiting time improvement factor, is greater when the system is not heavily loaded. (In all of the tests, the mean waiting time improvement factor was at least 50% when the average CPU utilization was less than 0.50.) In systems with partially replicated data or non-uniform query arrival rates, the improvements were even more significant. While the throughput increase was small in cases with fully replicated data and uniform workloads, the maximum throughput improvement factor was actually in the range of 10-30% (along with a significant mean waiting time improvement factor) in the cases with partial replication or non-uniform query arrival rates.

6.2. FUTURE RESEARCH DIRECTIONS

There are several issues related to load balancing that were not addressed in this thesis. First, this study has focused solely on the issue of how site load information can be used and the extent to which performance can be improved through its use. The problem of how to maintain this information was not addressed. Second, a problem with sender-initiated load balancing schemes is that a number of query units could be assigned to a lightly loaded (or idle) site simultaneously by more than one site, thus overloading the site [Livn83] [Eage85]. This problem was also not addressed here. An important topic for future research is the design of an information policy and a control policy that will not overburden either the sites or the communications subnetwork, and yet will provide the sites with information that is sufficiently current so that the performance improvements provided in this study are not lost. Approperiate low-

level network support, as offered by ATON design [Livn85a], could be helpful here.

Another obvious area for future work would be to implement the load-balanced query processing approach, including the static planning, dynamic allocation and refining phases, in a real (or testbed) locally distributed database system. Evaluating the complete approach using simulation methods would require a very detailed simulation model, including a more general model of query structures and processing sequences. Even if such a model were developed, important implementation details (such as the overheads due to the dynamic allocation algorithm or to maintaining load information) would probably be quite difficult to model convincingly.

There are several possible research directions related to the proposed load balanced query processing approach. First, a detailed comparison between the plans generated by algorithm LBQP and by existing query processing algorithms such as that of System R^* would be helpful to further validate the heuristics used in the static planing phase (i.e., to investigate the static optimality of the plans generated by LBQP). Second, logical plans in LBQP must be linear sequences of query units, and communications are restricted to being between adjacent query units. It would be worthwhile extending the dynamic query unit algorithms for use on more general query plans. These algorithms could then also be applied to solving more general task allocation problems for distributed computing systems. Third, algorithm LBQP was developed for use in local area networks, so it would be interesting to see how it might be extended for use in systems with geographically distributed data (and what performance improvements might be obtained by doing so).

Another possible generalization of the dynamic query allocation algorithms would be to extend them so that another resource, buffer space, is considered. Buffer space is a valuable resource that will certainly affect the performance of a query processing plan. At a heavily loaded site, not only do long queues form at the CPU and I/O service centers, causing performance to degrade, but also less buffer space is likely to be available for each query. Therefore, it may be important to consider the buffer space requirements of query processing plans and the buffer space available at candidate processing sites.

Since this thesis has considered only read-only queries, another topic for future work would be to investigate how updates will affect the load-balanced query processing algorithm. It is unlikely that updates will necessitate changes in the proposed algorithm, at least for systems which read-lock one copy and write-lock all copies of replicated data items (like System R* [Ceri84]). In such systems, the overhead for locking and updating replicated data is the same for all copies of an item. What will change when updates are considered, however, is the degree to which increasing the number of copies can help to improve performance for a given workload — the gains due to load balancing and to local accessibility of data have to be traded off against increased update costs for the data. Finally, it would be interesting to see how the proposed load-balanced query processing algorithm could be put to use in environments where high availability is an issue. Since LBQP dynamically maps a compiled query to its execution sites at runtime, considering a set of candidate sites for each query unit, LBQP would simply need to be informed of the available sites in order to execute a query in a partially operational system.

REFERENCES

[Alon84] R. Alonso, Query optimization in distributed database management systems through load balancing, talk presented at the University of Wisconsin-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), 57-68.

[Bern79] P. A. Bernstein and N. Goodman, Full reducers for relational queries using multi-attribute semi- joins, *Proceedings of the 1979 NBS Symposium on Computer Networks*, December 1979.

[Bern81a] 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 Database Systems,* 6, 4 (December 1981), 602-625.

[Bern81b] P. A. Bernstein and D. W. Chiu, Using semijoins to solve relational queries, *Journal of the ACM,* 28, 1 (January 1981), 25-40.

[Bitt83] D. Bitton, D. J. DeWitt and C. Turbyfill, Benchmarking database systems: a systematic approach, *Proceedings of the 9th International Conference on Very Large Data Bases*, October 1983.

[Blac82] P. A. Black and W. S. Luk, A new heuristic approach for generating semi-join programs for distributed query processing, *IEEE COMPASAC*, 1982.

[Blas76] M. W. Blasgen and K. P. Eswaran, On the evaluation of queries in a relational data base system, *IBM Research Rep. RJ1745*, April 1976.

[Blas77] M. W. Blasgen and K. P. Eswaran, Storage and access in relational data bases, *IBM Systems Journal*, 1977, 363-377.

[Bokh79] S. H. Bokhari, Dual processor scheduling with dynamic reassignment, *IEEE Transactions on Software Engineering,* SE-5, 4 (July 1979), 341-349.

[Brya81] R. Bryant and R. Finkel, A stable distributed scheduling algorithm, *Proceedings of the 2nd International Conference on Distributed Computing Systems*, April 1981 , 314-323.

[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*, May 1985.

[Ceri84] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems,* McGraw-Hill Publishing Co., 1984.

[Chan82] J. M. Chang, *Query processing in a fragmented database environment*, Technical Report, Bell Laboratories, 1982.

[Cher83] D. R. Cheriton and W. Zwaenepoel, The distributed V kernel and its performance for diskless workstations, *Proceedings of the 10th Symposium on Operating Systems Principles*, 1983, 129-140.

[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.

[Chou85] H. T. Chou, D. J. DeWitt, R. H. Katz and A. C. Klug, Design and implementation of the Wisconsin storage system, *Software Practice and Experience* **15**, 10 (October 1985).

[Chow79] Y. C. Chow and W. H. Kohler, Models for dynamic load balancing in a heterogeneous multiple processor system, *IEEE Transactions on Computers*, May 1979, 354-361.

[Chu80] W. W. Chu, L. J. Holloway, M. T. Lan and K. Efe, Task allocation in distributed data processing, *IEEE Transactions on Computers*, **C-29**, 11 (November 1980), 57-69.

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

[Dani82] D. Daniels, P. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker and P. Wilms, An introduction to distributed query compilation in R*, *Proceedings of the Second International Conference on Distributed Databases* ,Berlin, September 1982.

[DeWi84a] D. J. DeWitt, R. H. Katz, L. D. Shapiro, M. Stonebraker and D. Wood, Implementation techniques for main memory database systems, *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, June 1984.

[DeWi84b] D. J. DeWitt, R. Finkel and M. Solomon, *The Crystal multicomputer: design and implementation experience*, Computer Science Technical Report #553, Computer Sciences Department, University of Wisconsin-Madison, September 1984.

[Eage84] D. L. Eager, E. D. Lazowska and J. Zahorjan, *Dynamic load sharing in homogeneous distributed systems*, Technical Report 84-10-1, University of Washington, October 1984.

[Eage85] D. L. Eager, E. D. Lazowska and J. Zahorjan, *A comparison of receiver-initiated and sender-initiated dynamic load sharing*, Technical Report 85-04-1, University of Washington , April 1985.

[ElDe78] O. I. El-Dessouki, *Program partitioning and load balancing in network computers,* Ph.D. Dissertation, Illinois Institute of Technology, December 1978.

[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, 169-180.

[Epst80a] R. Epstein and M. Stonebraker, Analysis of distributed database processing strategies, *Proceedings of the 6th International Conference on Very Large Data Bases*, October 1980, 82-101.

[Epst80b] R. Epstein, *Query processing in a distributed data base environment*, Ph.D. Dissertation, University of California, Berkeley, 1980.

[Ferr85] D. Ferrari, *A study of load indices for load balancing schemes*, University of California, Berkeley, 1985.

[Good79] N. Goodman, P. A. Bernstein, E. Wong, C. L. Reeve and J. J. B. Rothine, *Query processing in SDD-1*, Technical Report, CCA-79-06, October 1979.

[Goud81] M. G. Gouda and U. Dayal, Optimal semijoin schedules for query processing in local distributed database systems, *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, April 1981 , 164-175.

[Gyly76] V. B. Gylys and J. A. Edqards, Optimal partitioning of workload for distributed systems, *Digest of Papers, COMPCON Fall 1976*, September 1976, 353-357.

[Hamm80] M. M. Hammer and D. W. Shipman, Reliability mechanism for SDD-1: A system for distributed databases, *ACM Transactions on Database Systems*, **5**, 4 (December 1980), 431-466.

[Hevn79] A. R. Hevner and S. B. Yao, Query processing in distributed database systems, *IEEE Transactions on Software Engineering*, **SE-5**, 3 (May 1979), 177-187.

[Hevn80] A. R. Hevner, *The optimization of query processing in distributed database systems*, Ph.D. Dissertation, Department of Computer Science, Purdue University, Lafayette, Indiana., 1980.

[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, 151-160.

[Kers82] L. Kerschberg, P. D. Ting and S. B. Yao, Query optimization in star computer networks, *ACM Transactions on Database Systems*, 7, 4 (December 1982), 678-711.

[Liu82] A. C. Liu and S. K. Chang, Site selection in distributed query processing, *Proceedings of the 3rd International Conference on Distributed Computing Systems*, Miami/Ft. Lauderdale, Florida, October 1982, 7-12.

[Livn82] M. Livny and M. Melman, Load balancing in homogeneous broadcast distributed systems, *Proceedings of ACM Computer Network Performance Symposium*, April 1982, 47-55.

[Livn83] M. Livny, *The study of load balancing algorithms for decentralized distributed processing systems*, Ph.D. Dissertation, Weizmann Institute of Science, August 1983.

[Livn85] M. Livny, *DENET — a Modula-2 based simulation language,* University of Wisconsin-Madison, (in preparation).

[Livn85a] M. Livny and U. Manber, Distributed computation via active messages, *IEEE Transactions on Computers,* C-34, 12, (December 1985), 1185-1190.

[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, et. al. (editors), Springer-Verlag Berlin/Heidelberg, 1985, 31-47.

[Ma82] P. R. Ma, E. Y. S. Lee and M. Tsuchiya, A task allocation model for distributed computing systems, *IEEE Transactions on Computers,* C-31, 1, January 1982.

[McCo81] R. McCord, Sizing and data distribution for a distributed database machine, *Proceedings of the ACM-SIGMOD International Confer-*

ence on Management of Data, May 1981.

[Melm84] M. Melman and M. Livny, The DISS methodology of distributed system simulation, *Simulation*, April, 1984.

[Ni81] L. M. Ni and K. Hwang, Optimal load balancing strategies for a multiple processor system, *IEEE Conference on Parallel Processing*, 1981, 352-357.

[Ni81a] L. M. Ni and K. Abani, Nonpreemptive load balancing in a class of local area networks, *Proceedings of the 1981 Computer Networking Symposium*, December 1981, 113-118.

[Ni82] L. M. Ni, A distributed load balancing algorithm for point-to-point local computer networks, *Proceedings of COMPCON*, Fall 1982, 116-123.

[Page83] T. W. J. Page, *Distributed query processing in local network databases*, Masters Thesis, University of California, Los Angeles, 1983.

[Pete83] J. L. Peterson and A. Silberschatz, *Operating System Concepts*, Addison-Wesley Publishing Company, 1983.

[Powe83] M. Powell and B. Miller, Process migration in DEMOS/MP, *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, 1983 , 110-119.

[Pric79] C. C. Price, *A nonlinear multiprocessor scheduling problem*, Ph.D. Dissertation, Texas A & M University, May 1979.

[Pric84] C. C. Price and S. Krishnaprasad, Software allocation models for distributed computing systems, *Proceedings of the 4th*

International Conference on Distributed Computing Systems, San Francisco, California, May 1984, 40-48.

[Rao79] G. S. Rao, H. S. Stone and T. I. Hu, Assignment of tasks in a distributed processor systems with limited memory, *IEEE Transactions on Computers,* **C-28,** 4, April 1979.

[Reis80] M. Reiser, S. S. Lavenberg, Mean-value analysis of closed multichain queuing networks, *Journal of the ACM,* **27,** 2 (June 1982), 313-322.

[Sacc82] G. M. Sacco and S. B. Yao, Query optimization in distributed database systems , in *Advances in Computers*, vol. 21 , New York, Academic Press, 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, 23-34.

[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.

[Smit75] J. M. Smith and P. Y. T. Chang, Optimizing the performance of a relational algebra database interface, *Communications the ACM,* **18,** 10 (October 1975), 568-579.

[Ston77a] H. S. Stone, Multiprocessor scheduling with the aid of network flow algorithms, *IEEE Transactions on Software Engineering,* **SE-3,** 1 (January 1977), 85-93.

[Ston77b] H. S. Stone, *Program assignment in three-processor systems and tricutset partitioning of graphs*, Technical Report No. ECE-CS-77-7, Department of Electrical Engineering, University of Massachusetts at Amherst, 1977.

[Ston78] H. S. Stone, Critical load factors in two processor distributed systems, *IEEE Transactions on Software Engineering*, **SE-4**, 3 (May 1978), 254-258.

[Wang85] Y. T. Wang and R. J. T. Morris, Load sharing in distributed systems, *IEEE Transactions on Computers*, **C-34**, 3 (March 1985), 204-217.

[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, 217-235.

[Wong80] E. Wong, Dynamic rematerialization: Processing distributed queries using redundant data, *Proceedings of the 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1980.

[Yao79] S. B. Yao, Optimization of query evaluation algorithms, *ACM Transactions on Database Systems*, **4**, 2 (June 1979), 133-155.

[YuCh83] 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, 30-39.

[Yu84] C. T. Yu and C. C. Chang, Distributed query processing, *Computing Surveys,* **16**, 4 (December 1984), 399-433.

[Yu85] C. T. Yu, C. C. Chang, M. Templeton, D. Brill and E. Lund, Query processing in a fragmented relational distributed system: Mermaid, *IEEE Transactions on Software Engineering,* **SE-11**, 8 (August 1985).

APPENDIX A

FORMAL MODELS FOR DYNAMIC QUERY UNIT ALLOCATION

With the definition of the unbalance factor of a system, the load-balanced dynamic query allocation problem defined in Section 4.3.3 can be formulated as a formal model. This appendix first presents a BNQ-based dynamic query unit allocation model. It is then extended to BNQRD-based dynamic query unit allocation.

A.1. Model for BNQ-Based Dynamic Query Unit Allocation

Let m be the number of query units of the query to be allocated, and let n be the number of sites in a locally distributed database system. The initial load at site s_j is N_j (the number of query units) for $1 \leq j \leq n$. Let variable X_{ij} be 1 when query unit q_i is assigned to site s_j, and otherwise be 0, where $1 \leq i \leq m$, $1 \leq j \leq n$. According to the definition of the BNQ-based load unbalanced factor (See Section 4.3.1), the UBF of the system can be expressed as:

$$UBF = \frac{\sum_{j=1}^{n}((N_j + \sum_{i=1}^{m} X_{ij}) - \bar{N})^2}{n}$$

where $\bar{N}$ is the average number of processes at each site after the query is allocated, which is:

$$\bar{N} = \frac{\sum_{j=1}^{n} N_j + m}{n}$$

Since:

$$\sum_{j=1}^{n}((N_j + \sum_{i=1}^{m} X_{ij}) - \bar{N})^2$$

$$= \sum_{j=1}^{n}((N_j - \bar{N})^2 + 2\cdot(N_j - \bar{N})\cdot\sum_{i=1}^{m} X_{ij} + (\sum_{i=1}^{m} X_{ij})^2)$$

$$= \sum_{j=1}^{n}(N_j - \bar{N})^2 + 2\cdot\sum_{j=1}^{n}((N_j - \bar{N})\cdot\sum_{i=1}^{m} X_{ij}) + \sum_{j=1}^{n}(\sum_{i=1}^{m} X_{ij})^2$$

$$= \sum_{j=1}^{n}(N_j - \bar{N})^2 + 2\sum_{j=1}^{n}(N_j \sum_{i=1}^{m} X_{ij}) - 2\bar{N}\sum_{j=1}^{n}\sum_{i=1}^{m} X_{ij} + \sum_{j=1}^{n}(\sum_{i=1}^{m} X_{ij})^2$$

$$= \sum_{j=1}^{n}(N_j - \bar{N})^2 + 2\sum_{j=1}^{n}(N_j \sum_{i=1}^{m} X_{ij}) - 2\cdot\bar{N}\cdot m + \sum_{j=1}^{n}(\sum_{i=1}^{m} X_{ij})^2$$

and the first and the third terms are constants, the objective of minimizing the UBF is then to minimize:

$$2\sum_{j=1}^{n}(N_j \sum_{i=1}^{m} X_{ij}) + \sum_{j=1}^{n}(\sum_{i=1}^{m} X_{ij})^2 \qquad (A.1)$$

The total communications cost assuming that each pair of query units q_i and q_{i+1} is executed at different sites will be $\sum_{i=0}^{m} C_i$. The pairs of query units q_i and q_{i+1} which are allocated at the same site cause no communications cost, and therefore the total communications cost of an allocation plan can be expressed as:

$$\sum_{i=0}^{m} C_i - \sum_{j=1}^{n}\sum_{i=0}^{m}(C_i \cdot X_{ij} \cdot X_{(i+1)j}) \qquad (A.2)$$

The constraints of the query unit allocation problem are:

(1) A query unit is only allocated to one site. That is:

$$\sum_{j=1}^{n} X_{ij} = 1 \qquad \text{for } 1 \le i \le m; \tag{A.3}$$

(2) A query unit can only be allocated to the sites that are in its feasible assignment set. That is:

$$X_{ij} = \begin{cases} 0, 1 & \text{for i, j where } s_j \text{ is in the feasible assignment set of } q_i \\ 0 & \text{otherwise} \qquad (1 \le i \le m, 1 \le j \le n) \end{cases} \tag{A.4}$$

Thus, the BNQ-based allocation problem is to find the solutions for X_{ij} which minimize (A.1) and (A.2) subject to the constraints (A.3) and (A.4).

A.2. Model for BNQRD-Based Allocation

In BNQRD-based query unit allocation, each query unit is classified as being either I/O-bound or CPU-bound. BNQRD-based load balancing takes place among the same type of query units. Let X_{ij} be 1 if q_i is an I/O-bound query and it is allocated to site s_j, and otherwise be 0. Let Y_{ij} be 1 if q_i is a CPU-bound query and it is allocated to site s_j, and otherwise be 0. The BNQRD-based unbalance factor has two components, one related to I/O-bound query units and the other related to CPU-bound query units. The objective of minimizing the unbalance factor can be decomposed into three objectives of minimizing these two components. For each component, the method used in deriving the objective function for BNQ-based allocation (i.e., A.1) described in the last section can be used. The objective function of minimizing the BNQRD-based unbalance factor can thus be expressed as:

$$2 \sum_{j=1}^{n} (N_{IO_j} \sum_{i=1}^{m} X_{ij}) + \sum_{j=1}^{n} (\sum_{i=1}^{m} X_{ij})^2$$

$$+ 2 \sum_{j=1}^{n} (N_{CPU_j} \sum_{i=1}^{m} Y_{ij}) + \sum_{j=1}^{n} (\sum_{i=1}^{m} Y_{ij})^2 \qquad (A.5)$$

where N_{IO_j} and N_{CPU_j} are the number of I/O-bound query units and the number of CPU-bound query units at site s_j before the query is allocated, respectively.

Since any two adjacent query unit, q_i and q_{i+1} (whether I/O-bound or CPU-bound) will eliminate some communications cost, the total communications cost can be expressed as:

$$\sum_{i=0}^{m} C_i - \sum_{j=1}^{n} \sum_{i=0}^{m} (C_i \cdot (X_{ij} + Y_{ij}) \cdot (X_{(i+1)j} + Y_{(i+1)j})) \qquad (A.6)$$

The constraints in this case are:

(1) Each query unit can only be assigned to one site. That is

$$\sum_{j=1}^{n} X_{ij} = \begin{cases} 1, & \text{if } q_i \text{ is I/O-bound} \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } 1 \le i \le m \qquad (A.7)$$

and

$$\sum_{j=1}^{n} Y_{ij} = \begin{cases} 1, & \text{if } q_i \text{ is CPU-bound} \\ 0 & \text{otherwise} \end{cases} \qquad \text{for } 1 \le i \le m \qquad (A.8)$$

(2) Each query unit can only be allocated to the sites that are in its feasible assignment set:

$$X_{ij} = \begin{cases} 0, 1 & \text{for i, j where } q_i \text{ is I/O-bound and} \\ & \quad s_j \text{ is in the feasible assignment set of } q_i \\ 0 & \text{otherwise} \qquad (1 \le i \le m,\ 1 \le j \le n) \end{cases} \qquad (A.9)$$

$$Y_{ij} = \begin{cases} 0, 1 & \text{for i, j where } q_i \text{ is CPU-bound and } s_j \text{ is in the feasible assignment set of } q_i \\ 0 & \text{otherwise} \end{cases} \quad (1 \leq i \leq m,\ 1 \leq j \leq n) \qquad (A.10)$$

The BNQ-based dynamic query unit allocation problem is to find the solutions for X_{ij} and Y_{ij} which minimize (A.5) and (A.6) subject to the constraints (A.7) and (A.8).

It should be pointed out that the discussion in this appendix is only to illustrate that the dynamic query unit allocation problems can indeed be formulated as integer programming problems. Issues such as using the minimum number of variables to reduce the problem size are not considered here.

APPENDIX B

SIMULATION RESULTS OF CHAPTER 2

In the simulations described in Chapter 2, the main metric measured was the mean waiting time of queries. This appendix presents these results with their confidence intervals obtained at the 95% confidence level.

Mean Waiting Time versus think_time				
think_time	Allocation Algorithm			
	LOCAL	BNQ	BNQRD	LERT
4.00	1.6653 ±0.0382	1.6698 ±0.0422	1.4843 ±0.0410	1.5148 ±0.0404
6.00	0.9591 ±0.0295	0.7872 ±0.0277	0.6755 ±0.0222	0.6662 ±0.0271
8.00	0.6000 ±0.0185	0.4092 ±0.0122	0.3776 ±0.0129	0.3668 ±0.0101
10.00	0.4356 ±0.0124	0.2675 ±0.0083	0.2413 ±0.0072	0.2379 ±0.0067
12.00	0.3475 ±0.0077	0.1855 ±0.0039	0.1697 ±0.0044	0.1630 ±0.0036
14.00	0.2811 ±0.0060	0.1356 ±0.0033	0.1208 ±0.0028	0.1246 ±0.0028
16.00	0.2313 ±0.0052	0.1060 ±0.0024	0.0990 ±0.0025	0.0954 ±0.0024
18.00	0.1996 ±0.0047	0.0840 ±0.0021	0.0776 ±0.0018	0.0753 ±0.0022

Table B.1: Chapter 2: Mean Waiting Time vs. *think_time*

Mean Waiting Time versus think_time				
mpl	Allocation Algorithm			
	LOCAL	BNQ	BNQRD	LERT
5	0.0601 ± 0.0018	0.0152 ± 0.0005	0.0148 ± 0.0004	0.0156 ± 0.0004
10	0.1507 ± 0.0038	0.0603 ± 0.0018	0.0572 ± 0.0015	0.0580 ± 0.0018
15	0.2707 ± 0.0061	0.1410 ± 0.0037	0.1299 ± 0.0031	0.1294 ± 0.0030
20	0.4356 ± 0.0124	0.2675 ± 0.0083	0.2413 ± 0.0072	0.2379 ± 0.0067
25	0.6606 ± 0.0149	0.4406 ± 0.0114	0.3976 ± 0.0093	0.3952 ± 0.0106
30	0.9612 ± 0.0259	0.7187 ± 0.0240	0.6596 ± 0.0201	0.6232 ± 0.0180
35	1.3783 ± 0.0365	1.2106 ± 0.0431	1.0301 ± 0.0357	1.0244 ± 0.0349
40	1.9577 ± 0.0450	1.8848 ± 0.0634	1.7033 ± 0.0561	1.5494 ± 0.0388
45	2.7475 ± 0.0647	2.9407 ± 0.0749	2.5761 ± 0.0836	2.6114 ± 0.0709

Table B.2: Chapter 2: Mean Waiting Time vs. *mpl*

Mean Waiting Time versus think_time				
num_sites	Allocation Algorithm			
	LOCAL	BNQ	BNQRD	LERT
2	0.4350 ± 0.0099	0.3472 ± 0.0086	0.3244 ± 0.0074	0.3313 ± 0.0085
4	0.4345 ± 0.0101	0.2876 ± 0.0069	0.2597 ± 0.0068	0.2595 ± 0.0066
6	0.4356 ± 0.0124	0.2675 ± 0.0083	0.2413 ± 0.0072	0.2379 ± 0.0067
8	0.4418 ± 0.0095	0.2522 ± 0.0066	0.2306 ± 0.0055	0.2269 ± 0.0054
10	0.4478 ± 0.0091	0.2506 ± 0.0059	0.2193 ± 0.0051	0.2144 ± 0.0047
12	0.4418 ± 0.0084	0.2479 ± 0.0048	0.2199 ± 0.0047	0.2154 ± 0.0043
14	0.4378 ± 0.0084	0.2435 ± 0.0046	0.2131 ± 0.0041	0.2135 ± 0.0040

Table B.3: Chapter 2: Mean Waiting Time vs. *num_sites*

Mean Waiting Time versus think_time				
$class_{io}\_prob$	Allocation Algorithm			
	LOCAL	BNQ	BNQRD	LERT
0.2	0.8596 ±0.0211	0.5470 ±0.0156	0.5196 ±0.0145	0.5307 ±0.0176
0.3	0.6652 ±0.0159	0.4229 ±0.0136	0.3859 ±0.0108	0.3843 ±0.0113
0.4	0.5443 ±0.0129	0.3259 ±0.0089	0.2977 ±0.0069	0.3007 ±0.0080
0.5	0.4356 ±0.0124	0.2675 ±0.0083	0.2413 ±0.0072	0.2379 ±0.0067
0.6	0.3697 ±0.0086	0.2210 ±0.0055	0.1961 ±0.0045	0.1913 ±0.0045
0.7	0.3315 ±0.0023	0.1796 ±0.0046	0.1665 ±0.0045	0.1606 ±0.0034

Table B.4: Chapter 2: Mean Waiting Time vs. $class_{io}\_prob$

Mean Waiting Time	
Estimation Error (%)	Mean Waiting Time
10	0.2348 ± 0.0058
20	0.2380 ± 0.0062
30	0.2406 ± 0.0057
40	0.2387 ± 0.0059
50	0.2472 ± 0.0064
60	0.2504 ± 0.0064
80	0.2757 ± 0.0074

Table B.5: Chapter 2: Mean Waiting Time vs. estimation error.

APPENDIX C

SIMULATION RESULTS OF CHAPTER 5

In the simulations described in Chapter 5, the main metric measured was the mean response time of queries. This appendix presents these results with their confidence intervals obtained at the 95% confidence level.

Fully Replicated Case (Expt. 1, Test 1)			
think_time	Mean Response Time		
	STATIC	LBQP	$RANDOM_F$
4.0	6.7386 ±0.2041	7.1209 ±0.2623	9.4899 ±0.4003
8.0	5.1497 ±0.2707	4.9445 ±0.3117	7.2437 ±0.4109
12.0	4.4656 ±0.2057	3.8217 ±0.1695	5.6299 ±0.3068
16.0	4.2842 ±0.1781	3.5854 ±0.1500	4.7978 ±0.1903
22.0	3.8597 ±0.1573	3.1107 ±0.0912	4.2321 ±0.2195
28.0	3.6976 ±0.1521	3.0932 ±0.1293	3.9427 ±0.1651

Table C.1: Chapter 5, Experiment 1, Test 1.

Partially Replicated Case (Expt. 1, Test 2)			
think_time	Mean Response Time		
	STATIC	LBQP	RANDOM$_P$
4.0	11.3396 ±0.3737	11.1704 ±0.4076	10.7460 ±0.4378
8.0	9.9673 ±0.3571	7.7476 ±0.3822	8.2205 ±0.4892
14.0	7.5296 ±0.4437	4.6288 ±0.2400	5.7520 ±0.3769
18.0	6.6118 ±0.4194	3.9213 ±0.1719	5.0247 ±0.2036
24.0	5.7023 ±0.2322	3.5293 ±0.1498	4.5756 ±0.2161
28.0	4.9018 ±0.3616	3.3131 ±0.1155	4.2037 ±0.1831

Table C.2: Chapter 5, Experiment 1, Test 2.

Partially Replicated Case 2 (Expt. 1, Test 3)			
think_time	Mean Response Time		
	STATIC	LBQP	RANDOM$_P$
4	23.0405 ±0.7207	21.2330 ±0.6368	21.5755 ±0.8444
8	22.6615 ±0.6605	16.7008 ±0.8677	17.8391 ±0.7444
12	21.2390 ±0.8454	13.2754 ±0.8220	14.8280 ±0.7981
16	19.6102 ±0.8879	9.9189 ±0.5737	11.3907 ±0.0502
20	17.7289 ±1.2021	7.3422 ±0.5032	9.0395 ±0.5603
24	15.1060 ±0.8855	5.7157 ±0.3573	7.3711 ±0.4813
28	13.4276 ±0.8519	4.9459 ±0.2431	6.1491 ±0.3962

Table C.3: Chapter 5, Experiment 1, Test 3.

Partially Replicated Case 3 (Expt. 1, Test 4)			
think_time	Mean Response Time		
	STATIC	LBQP	RANDOM$_P$
4	8.8340 ±0.3557	7.6707 ±0.2270	8.1832 ±0.3432
6	8.2374 ±0.5978	6.1862 ±0.3122	7.0149 ±0.3050
8	7.2378 ±0.4657	5.2651 ±0.2587	6.2027 ±0.3836
12	5.8048 ±0.3487	4.2391 ±0.1639	5.3105 ±0.2159
18	4.8491 ±0.2071	3.6541 ±0.1416	4.4598 ±0.1871
22	4.3868 ±0.2015	3.4449 ±0.1263	4.2118 ±0.2166
24	4.0508 ±0.2052	3.3506 ±0.1471	3.9057 ±0.2675
28	3.9102 ±0.1708	3.1799 ±0.1312	3.7816 ±0.1111

Table C.4: Chapter 5, Experiment 1, Test 4.

Fully Replicated Case (Expt. 2, Test 1)			
think_time	Mean Response Time		
	STATIC	LBQP	$RANDOM_F$
4	3.3408 ±0.1615	3.4835 ±0.2047	4.9231 ±0.3264
6	2.9487 ±0.1621	2.8015 ±0.1815	3.9547 ±0.2875
8	2.6017 ±0.1557	2.2342 ±0.1374	3.3880 ±0.2389
10	2.4144 ±0.1349	2.0897 ±0.0916	3.0390 ±0.1800
12	2.3549 ±0.1300	2.0137 ±0.0929	2.8435 ±0.2118
18	2.2491 ±0.0820	1.8117 ±0.0898	2.3356 ±0.0828

Table C.5: Chapter 5, Experiment 2, Test 1.

Partially Replicated Case 1 (Expt. 2, Test 2)			
think_time	Mean Response Time		
	STATIC	LBQP	$RANDOM_P$
2	6.8016 ±0.3744	4.9528 ±0.3235	6.2053 ±0.3643
6	4.7375 ±0.3256	2.7576 ±0.1487	3.8398 ±0.2568
10	3.5379 ±0.3221	2.1538 ±0.1028	2.9706 ±0.1517
14	3.1184 ±0.2240	2.0657 ±0.1162	2.5800 ±0.1613
18	2.5179 ±0.1586	1.9165 ±0.0792	2.2542 ±0.1323

Table C.6: Chapter 5, Experiment 2, Test 2.