Design of Distributed Databases on Local Computer Systems with a Multiaccess Network

BENJAMIN W. WAH, MEMBER, IEEE, AND YAO-NAN LIEN

Abstract—Concurrency control, distribution design, and query processing are some of the important issues in the design of distributed databases. In this paper, we have studied these issues with respect to a relational database on a local computer system connected by a multiaccess broadcast bus. A broadcast bus allows information to be distributed efficiently, and hence simplifies the solutions to some of these issues. A transaction model that integrates the control strategies in concurrency control and query processing is proposed. In concurrency control, the lock, unlock, and update of data are achieved by a few broadcasts. A dynamic strategy is used in query processing, as less data are transferred when compared to a static strategy. The status information needed in dynamic query processing can be conveniently obtained by broadcasting. Lastly, some NP-hard file placement problems are found to be solvable in polynomial time when updates are broadcast.

Index Terms—Broadcast, concurrency control, file allocation, local computer network, multiaccess bus, priority, query processing, transaction.

I. Introduction

ADVANCES in computer and communication technologies, coupled with the applications explosion, have led to the design of DDB's or *distributed database management systems.* A distributed database is a collection of data that belong logically to the same system, but are spread over multiple computers connected by a network [6]. Some of the key issues in designing DDB's are the distribution of data, the distributed processing of queries, the concurrency control of transactions, and the design of the supporting communication network.

Concurrency control maintains the integrity of information in a multiuser environment, and prevents updates by one user from interfering with retrievals and updates of another [3]. Concurrency control is complicated by the need to access information stored in many computers concurrently. Since communication delays usually prohibit instantaneous distribution of status information, transactions originating from different sites may interfere with each other. Further, a computer or the network may fail, and recovery must be possible.

Distribution design considers the partitioning of a database into fragments and the distribution of these fragments on the system [5]. The partition may be either horizontal, in which instances of an object are divided into possibly overlapping fragments, or vertical, in which attributes are divided into possibly overlapping fragments. The FAP, or (optimal) *file allocation problem*, distributes the fragments to satisfy system constraints, such as availability, reliability, and delay, and if the constraints and needs change dynamically, the fragments are allowed to migrate [9], [12], [26]. A special case of the FAP is the SFAP, or *simple file allocation problem*, which considers the allocation of multiple copies of a single file, and the effects of queries, updates, and data storage are represented as costs in the system [21].

The operation performed on the data in a database is called a *query*. In a DDB, a query may originate from any site in the system. In processing a query, either the required files are assembled at a single site and the query processed there, or the query and the intermediate results are sent and processed sequentially through the sites containing the files. A combination of the two strategies is also possible. The design of an effective strategy of local data processing and data transmissions to minimize the amount of data transfer or the response time of processing is known as the DQP, or *distributed query processing problem* [1].

The relationships among the above control issues of DDB's are shown in Fig. 1. For example, the files are partitioned and placed according to the characteristics of the network and the query; the design of the query processing strategy, the concurrency control algorithm, and the communication network depend on the placements and partitioning of the files. The issues shown in Fig. 1 are also interrelated to other important problems, such as failure recovery, logical database design, and directory management. It is very difficult to solve these problems as a whole, so the designer usually decomposes them into independent problems and studies each based on simplified assumptions of others.

One potential problem of the above approach is that oversimplification may lead to undesirable results. An example is shown in the use of a network model that is independent of the hardware characteristics. A simple network model is often used because, besides simplifying the design of DDB strategies, it allows the software to be transportable, and permits many applications of different requirements to share the same network. A protocol hierarchy is used, which allows the DDB to interact with the lower levels through system calls. As a result, the status information needed in the control and the interchange of data among sites must be formulated into messages that are recognized by the lower levels of the hierarchy. This mismatch between the characteristics of the physical network and the requirements of the DDB results in inefficiency

Manuscript received June 1, 1984. This research was supported in part by the National Science Foundation under Grant ECS81-05968 and by CIDMAC, a research unit of Purdue University, sponsored by Purdue, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation, and TRW.

The authors are with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.

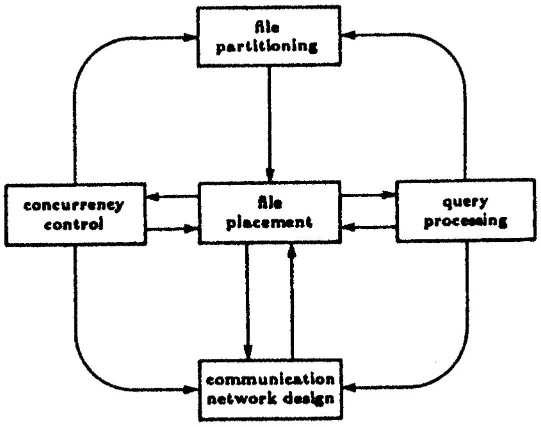


Fig. 1. The relationships of some design issues in distributed databases. (An arrow from Problem A to Problem B indicates either the provision of information from A to B, or the assumption of a solution for A in solving B.)

and increased complexity of the control strategies. To improve the performance, the capabilities of the network must be taken into account in the design.

In this paper, we have studied the design of a DDB based on the characteristics of a *local computer network*. The network is assumed to be a reliable *multiaccess bus* with the *broadcast capability*, such as the Ethernet [23]. Multiple computers contend for the use of the bus, and contention is resolved by a distributed contention-resolution algorithm. The broadcast capability eliminates many duplicate messages in the network. It is helpful, but not essential, to assume that messages have priorities, and the network always transmit a message of higher priority first. The fact that a relatively slow bus is used means that the communication overhead is proportional to the amount of data sent, and the fraction of bandwidth that each computer has in communicating with others is small as compared to the local processing speed when the number of computers is large. The local processing overhead is, therefore, assumed to be negligible.

The data model is assumed to be relational, in which a relation is the basic unit to be stored, locked, and queried [11]. A request is defined in terms of a transaction, which is a sequence of queries, each in the form of a conjunctive equijoin with projections, selections, and joins. The order of execution of queries in a transaction is assumed to be fixed, but the order of equijoins in each query is flexible. Queries are assumed to originate from sites, and query rates of each relation are known. The directory for locating relations is assumed to be fully redundant at each site.

In this paper, the query processing and concurrency control problems are studied in an integrated fashion, although they are usually treated as independent problems elsewhere. This integrated study allows the effects of locking and unlocking to be taken into account in the query processing strategies, and vice versa. The following transaction processing sequence is defined and is illustrated in Fig. 2 for a transaction with three queries.

1) Initial Processing Phase: A new transaction is processed at the originating site. The set of sites containing relations that are referenced by the transaction (*related sites*) are identified. A related site is excluded if it contains relations that are a subset of relations in another related site. Each query in the transaction is then restructured into a form that can be processed efficiently.

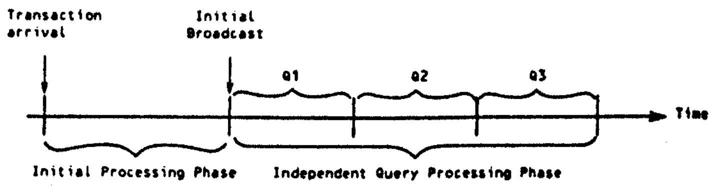


Fig. 2. The processing sequence of a transaction with three queries Q_1, Q_2, Q_3.

2) Initial Broadcast Phase: The restructured transaction is broadcast to all sites. This distributes the transaction to the related sites and provides information for concurrency control to other sites. Each site records the necessary information into a *Broadcast Transaction Table*. The time of the initial broadcast is called the *Initial Broadcast Time*. Initial broadcasts should be given higher priorities to transmit to allow transactions to be distributed to all sites as early as possible.

3) Independent Query Processing Phase: Since the processing order of queries in a transaction is assumed to be fixed, each query is processed in sequence by all related sites after the initial broadcast. Strategies for DQP are presented in Section III. The consistency of queries is maintained by the concurrency control algorithm and is discussed in Section IV.

In the remaining sections, we will show how the characteristics of a local broadcast bus can be effectively used to improve the performance of a DDB.

II. File Allocation

In this section, some simplified versions of the FAP are studied. It is assumed that file allocation is done at design time and that the average access rates to files from each site can be estimated; hence the problem can be solved independently. By simplifying the FAP and investigating the limitations of these problems, the algorithms obtained are useful as heuristics to solve the general problems.

A. Simple File Allocation

The SFAP considers the allocation of a single file, in which the effects of queries, updates, and data storage are represented as costs [21]. It has been shown to be NP-hard [13], [14], which means that the computation time for all known optimal algorithms for this class of problems increases at least exponentially with the problem size. Numerous exhaustive optimal algorithms and suboptimal heuristics have been studied [12], [26].

In this section, a solution to the SFAP for local multiaccess networks is proposed. Since only one broadcast is required to either lock, unlock, or update a file, the problem is found to be polynomially solvable with respect to the number of sites. The general notation used in the formulation are as follows.

N	Number of sites in the system.
F	Set of files in the database, $\lvert F \rvert = m$.
I^f	Index set of sites with a copy of file f.
λ_j^f	Query load originating at site j for file f per unit time.
ϕ_j^f	Update load originating at site j for file f per unit time.

d	Cost per unit size of query from any site to any other site.
d'	Cost per unit size of update from any site to all other sites.
σ_k^f	Storage cost per unit time of file f at site k.
L^f	Length of file f.
S_k	Storage capacity at site k.

$$Y_k^f = \begin{cases} 1 & \text{if file } f \text{ exists at site } k \\ 0 & \text{otherwise.} \end{cases}$$

Since the per-unit cost of access is constant in a broadcast bus, the query and update costs of a file are independent of the placement and the number of copies of the file.

In the following cost function for a given placement I, the index f is dropped because the problem is defined with respect to a single file.

$$C(I) = \sum_{j=1}^{N} \lambda_j d(1 - Y_j) + \sum_{j=1}^{N} \phi_j d' \left(1 - Y_j \prod_{\substack{k=1 \\ k \neq j}}^{N} (1 - Y_k)\right) + \sum_{k=1}^{N} \sigma_k Y_k \tag{1}$$

The first and last terms on the right-hand side of the above equation represent the query and storage costs, respectively. The second term accounts for the update costs. When multiple copies of a file exist in the system, each unit of update incurs a constant cost of d'. However, when only one copy of the file exists, updates originating from the site with the copy do not incur any cost because they do not have to be broadcast.

This problem can be formulated as an integer program with (1) as the cost function and constraints that at least one copy exists and Y_k is either 0 or 1. The solution to the problem will be simplified by considering the single-copy and the multicopy cases separately.

For the single-copy case, the cost function becomes

$$C(I) = \sum_{j=1}^{N} [(\lambda_j d + \phi_j d')(1 - Y_j) + \sigma_j Y_j]$$
$$= \sum_{j=1}^{N} (\lambda_j d + \phi_j d') + \sum_{j=1}^{N} (\sigma_j - \lambda_j d - \phi_j d') Y_j . \tag{2}$$

In addition to the fixed cost in (2), the cost of not placing the file at site j is $\lambda_j d + \phi_j d'$, while the cost of placing the file there is σ_j. Thus the file should be placed at site j if the cost difference $(\sigma_j - \lambda_j d - \phi_j d')$ there is minimum. The optimal allocation can, thus, be found in $O(N)$ time.

For the multicopy case, at least two copies must be allocated. The cost function is

$$C(I) = \sum_{j=1}^{N} [\lambda_j d(1 - Y_j) + \phi_j d'] + \sum_{k=1}^{N} \sigma_k Y_k$$
$$= \sum_{j=1}^{N} (\lambda_j d + \phi_j d') + \sum_{j=1}^{N} (\sigma_j - \lambda_j d) Y_j . \tag{3}$$

The first term in the above equation is constant. The optimal allocation can be found by allocating a copy at site j if $(\sigma_j - \lambda_j d)$ is negative. This will decrease the overall cost $C(I)$. In case that none or one of the cost differences is negative, two copies with the minimum cost differences are selected. The optimal allocation can hence be found in $O(N)$ time.

The global optimum is obtained by comparing the costs under the single-copy and multicopy cases. Therefore, the SFAP is optimally solvable in polynomial time. As an example, consider a system with four nodes, query rates $\lambda_* = [6,7,4,5]$, update rates $\phi_* = [3,6,2,4]$, and storage costs $\sigma_* = [3,5,2,5]$. The per-unit query and update costs, d and d', are assumed to be unity. In the single-copy case, $[\sigma_* - \lambda_* d - \phi_* d'] = [-6, -8, -4, -4]$. The single copy should be allocated at site 2 with cost 29. In the multicopy case, $[\sigma_* - \lambda_* d] = [-3, -2, -2, 0]$. Copies should be allocated to sites 1, 2, and 3 with a total cost 30. Comparing the single-copy and multicopy cases, a single copy should be placed at site 2.

The SFAP can also be solved efficiently when an availability constraint is included. Since the network is reliable, the availability constraint can be expressed into a minimum number of copies of the file, when the failure rates of sites are assumed to be identical. The locations for these copies can be determined as above.

The SFAP becomes NP-hard when an average-delay constraint is included. Given that the average delay of messages on the multiaccess bus is a constant c, the average delay for all queries and updates must satisfy

$$\sum_{j=1}^{N} [\lambda_j c(1 - Y_j) + \phi_j c] \leqslant D' \quad \text{or}$$
$$\sum_{j=1}^{N} \lambda_j c(1 - Y_j) \leqslant D. \tag{4}$$

In (4), updates are assumed to be always broadcast. The SFAP with the above average-delay constraint can also be separated into the single-copy and multicopy cases. The single-copy case is solvable in $O(N)$ time. For the multicopy case, the cost formula in (3) can be rewritten as

$$C(I) = \sum_{j=1}^{N} (\phi_j d' + \sigma_j) + \sum_{j=1}^{N} (\lambda_j d - \sigma_j)(1 - Y_j).$$

Since the first term on the right-hand side is constant, the optimization problem for the multicopy case can be written as

$$\text{maximize } C(I) = \sum_{j=1}^{N} (\sigma_j - \lambda_j d)(1 - Y_j) \tag{5}$$

$$\text{subject to } \sum_{j=1}^{N} \lambda_j (1 - Y_j) \leqslant \frac{D}{c}$$

and

$$(1 - Y_j) = 0 \text{ or } 1.$$

This problem is reducible from the 0-1 knapsack problem[1] with N objects by the variable transformation $Z_j = 1 - Y_j$,

[1]Given a finite set U of m objects and a knapsack of size B, there are a size $s(u) \in Z^+$ and a profit $p(u) \in Z^+$ for each $u \in U$, where Z^+ is the set of positive real numbers. The problem is to find a subset $U' \subseteq U$ that maximizes $\sum_{u \in U'} p(u)$ such that $\sum_{u \in U'} s(u) \leqslant B$.

where the profit and weight of the jth object are $(\sigma_j - \lambda_j d)$ and λ_j, respectively. However, the profits may be negative for some of the objects, and they are excluded from consideration in the optimization. The problem can be solved by dynamic programming algorithms or fully polynomial-time approximate schemes.

B. General FAP with Storage-Capacity Constraints

The general FAP considers the allocation of multiple files under such design requirements as delay, storage capacity, parallelism, and availability. As a simplification, each query is assumed to access a single file, and all file accesses are assumed to be independent. Only the storage capacities of sites are considered in the following formulation.

Using the notation defined in the last section, the problem with storate-capacity constraints can be formulated as

$$\text{minimize } C(I^1, \cdots, I^{|F|}) = \sum_{f,j} \lambda_j^f d(1 - Y_j^f) + \sum_{f,j} \phi_j^f d' \left(1 - Y_j^f \prod_{k \neq j} (1 - Y_k^f)\right) + \sum_{f,j} \sigma_j^f Y_j^f \tag{6}$$

subject to

a) $Y_j^f = 0 \text{ or } 1, \quad j = 1, \cdots, N, f \in F$

(each site contains at most one copy of each file);

b) $\sum_{j=1}^{N} Y_j^f \geq 1, \quad f \in F$

(at least one copy of each file exists in the system);

c) $\sum_{f \in F} L^f Y_j^f \leq S_j, \quad j = 1, \cdots, N$

(the storage capacity at each site is not exceeded).

The above problem is nonlinear, but can be linearized easily by using different index variables [15]. This problem is NP-hard as shown in the following theorem.

Theorem 1: The optimization problem defined by (6) is NP-hard.

Proof: We show that the 0-1 knapsack problem reduces to this problem in polynomial time. Given an instance of the 0-1 knapsack problem, an instance of the problem in (6) can be formed with the following parameters: $N = 2$; $F = U$; $L^f = s(f)$ for $f \in F$; $S_1 = \Sigma_{f \in F} L^f$; $S_2 = B$; $\sigma_1^f = 0$ for $f \in F$; λ_1^f = a large constant for $f \in F$; ϕ_1^f, λ_2^f, and σ_2^f are chosen such that $\lambda_2^f d - \phi_1^f d' - \sigma_2^f = p(f)$ for $f \in F$. Basically, the first site has large enough capacity to hold a copy of all files in F. The query and storage costs are chosen such that a copy of all files are allocated at site 1. Therefore, the problem becomes the packing of the second site, which is the standard knapsack problem. ∎

Although the problem in (6) resembles the packing of $m \cdot n$ copies into m knapsacks, there is a subtle difference here. The profit of allocating the first copy of file f to site j is $(\lambda_j^f d + \phi_j^f d' - \sigma_j^f)$, while the profit of allocating the second copy of file f to site k, $k \neq j$, is $(\lambda_k^f d - \phi_j^f d' - \sigma_k^f)$. The extra term $\phi_j^f d'$ in the profit of the second copy offsets the additional profit incurred when the first copy is allocated. The profit of allocating other copies of file f to site l, $l \neq j$ and $l \neq k$, is $(\lambda_l^f d - \sigma_l^f)$. Note that these profits may be negative. Due to the above and since one or more copies must exist in the system, the problem is decomposable into multiple independent knapsack problems only if at least two copies have been allocated in the system. The overall profit can then be maximized by independently optimizing the profit of each site with the remaining capacity. An optimal algorithm, therefore, consists of enumerating the allocations of the first two copies of each file, and solving N knapsack problems for each combination. The structure of the state-space tree is as follows.

1) The root is in level 0.

2) In the first m levels, the first copy of all files are allocated. Level f, $1 \leq f \leq m$, represents the allocation of file f. Each file can be allocated to one of the N sites. Thus the degree of the tree in the first m levels is N. If a file cannot fit in a site, the subtree representing this allocation is considered infeasible and terminated.

3) The allocation of the second copy of each file is carried out in the $(m + 1)$th to the $2m$th levels of the tree. Since the second copy must not be allocated to the site containing the first copy and may not have to be allocated in the DDB, the degree of branching at each level is N. If a copy cannot fit in a site, the subtree representing this allocation is considered infeasible and terminated. Further, if an allocation of the second copy at site k results in negative value of $(\lambda_k^f d - \sigma_k^f)$, then the second copy should not be allocated at site k, and the corresponding subtree is also terminated.

4) For all active nodes left in the state-space tree in level $2m + 1$, the allocation of the remaining copies is solved as N single-knapsack problems. For a particular site, the files to be considered for packing into the remaining capacity are those that have not been allocated at this site and those with the second copy allocated in the system. The algorithm for solving the conventional 0-1 knapsack problem has to be modified to take into account the negative profits.

Bounding criteria similar to those used in branch-and-bound algorithms can be developed here [15]. In particular, an upper bound on the profit can be computed for each node in the state-space tree by a linear program or a greedy algorithm without the integrality constraints. The maximum profit of the currently available feasible solutions is kept in the incumbent. If the upper bound of a node is smaller than the incumbent, then the subtree originating from this node is pruned because it cannot possibly lead to an optimal solution. An example illustrating the allocation of two files in a system with two sites using a depth-first search is shown in Fig. 3. In computing the upper bound for a node in the branch-and-bound tree, each site is treated as an independent knapsack. Given that i, $i = 0, 1, 2$, copies of a file to be included in the knapsack have been allocated in ascendant levels of this node, the profit of the file to be used is that of the $(i + 1)$th copy. A greedy algorithm computes the upper bound by assigning the files to each knapsack in descending ratios of profit to length until the knapsack is full. To keep the knapsack full, the last

Cost of query, update, and storage

	file 1		file 2	
	site 1	site 2	site 1	site 2
λ_j^d	8	9	6	7
$\varphi_j^{d'}$	1	1	1	1
σ_k	4	6	4	6

Storage limit of sites

site 1	site 2
5	6

Length of files

file 1	file 2
3	4

Profits of first and second copies

	file 1		file 2	
	site 1	site 2	site 1	site 2
First copy	5	4	3	2
Second copy	3	2	1	0

Allocations:

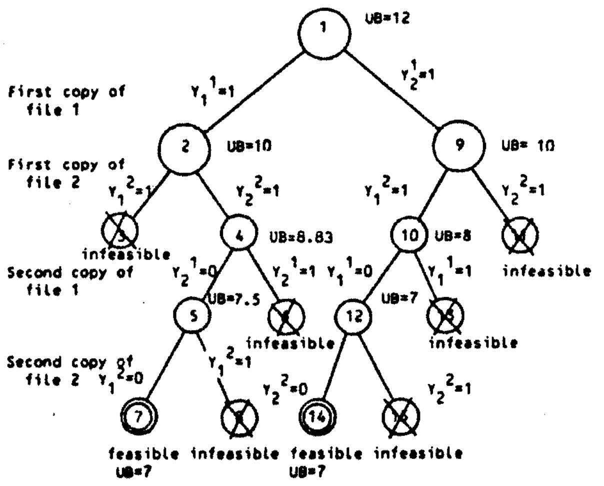


Fig. 3. Example to illustrate the general file allocation problem with capacity constraints ($N = 2$; $|F| = 2$; the number in each node indicates the search order; UB is the upper bound as solved by a greedy algorithm).

file may be partially allocated. The worst-case number of knapsack problems to be solved in this approach is N^{2m}, which is much better than an exhaustive enumeration.

In summary, solution algorithms for the SFAP are efficient and can be evaluated in either polynomial or pseudo-polynomial time. These algorithms can be applied in real time when the access rates change. However, optimal algorithms for the general FAP are complex and should be used in the design stage only.

III. Distributed Query Processing

A. Previous Work

The processing of a query involves the identification of the set of physical fragments referenced by the query (*materialization*), the selection of the order of execution of operations, and the selection of the method for executing each operation. In previous studies, a materialization is assumed for each query, the order of processing is optimized, and the operations are clustered into local actions. Materialization is assumed to be nonredundant, in which one copy of each fragment is identified in processing the query. Further, the transmission cost per unit data is assumed to be constant for any two points in the network, and the cost of local processing is negligible. Under these circumstances, the query processing problem is simplified to the estimation of the data statistics and the optimization of the transmission order, such that the average total data transmission or completion time is minimized [8].

Most of the previous studies on distributed query processing were based on relational databases in a nonbroadcast system. Since the problem is NP-hard, heuristic algorithms were generally used [28]. Bernstein and Chiu incorporated semijoins in DQP strategies and showed that they were more efficient than full joins [2], [4]. They observed that when two distributed relations are joined, one of the joining relations can be reduced in size by deleting those tuples that do not appear in the final result. This may be achieved by sending values of the joining attribute of the second relation to the first one and performing a join there. This reduction effect is called a *join restriction* [29]. The join is completed by sending the reduced form of the first relation to the second one. When multiple relations are to be joined, all semijoins are scheduled first. The remaining fragments are then transferred to a single site at which the complete join is performed. Since the major overhead is due to semijoins, the minimization of this overhead is the objective in most studies. A comprehensive approach of query processing that incorporates semijoins has been developed by Apers, Hevner, and Yao [1], [17], [29].

Studies have been made to improve semijoins. Yu *et al.* reduced the transmission cost by incorporating complement transmissions into semijoins [30]. The *complement* of an attribute is defined as the set of values absent from a given attribute. During semijoins, the complement of an attribute is sent if its cardinality is smaller than that of the original attribute. Nonetheless, there is no efficient algorithm to determine the optimal semijoin sequence, even for simple queries with one common joining attribute. Data compression techniques have also been applied in semijoins. For instance, a bit-vector indicating the absence or presence of a value can be sent instead of the attribute if the size of the bit-vector is smaller than the size of the attribute [16]. Suboptimal techniques for semijoins with smaller optimization overhead have also been studied [19]. Chen and Li employed a graphical approach to identify and eliminate redundant transmissions in a semijoin schedule [7].

Chu and Hurley have developed a unified approach that considers both the local processing and transmission costs [10]. A query-tree model for selecting the transmission sequence and the sites for executing a set of subqueries was proposed.

In short, previous studies on DQP have assumed that attributes are independent, ignored the effect of semijoins on nonjoining attributes, and used a nonredundant materialization as well as a static strategy. These do not always lead to good solutions for the following reasons. First, attributes in a real database are usually dependent, although the effect of semijoins on dependent attributes is difficult to estimate. Second, a semijoin on one attribute will affect other attributes. When relation $\boldsymbol{R}$ semijoins (equijoins) with relation $\boldsymbol{S}$ on one attribute, some tuples in $\boldsymbol{R}$ may be eliminated, which in turn may eliminate values in other attributes of $\boldsymbol{R}$. This is called an *indirect semijoin* and should be considered in optimizing the semijoin sequence. Third, redundant materialization has been found to result in less data transmission than nonredundant materialization [27]. Fourth, a static schedule is inflexible because a transaction cannot proceed once it is blocked by another transaction. On the other hand, a dynamic schedule allows relations that are not locked to be used first. Finally, the estimated statistics of the database are usually inaccurate for static strategies. The errors in estimation usually propagate regardless of the complexity of the model of estimation. Our study found that errors accumulate when a long sequence of semijoins are processed, and the use of unreliable statistics in optimization is unrewarding [27]. Moreover, the computational overhead of static strategies is high.

In the following sections, we describe a solution of the DQP with redundant materialization and a dynamic strategy. An attribute is first selected for broadcasting. The semijoins are performed and the statistics of the resulting database are collected at all sites. Based upon these statistics, the sites cooperatively select the next attribute to be broadcast. The process is then repeated. This strategy has been proposed before [22], [31], but was hampered by the high cost of collecting statistics after each semijoin. The use of a broadcast network allows the collection of statistics at a reasonable cost.

B. Query Processing Strategy

The processing of a query is divided into five phases.

1) Concurrency Control (CC) Phase: Our proposed concurrency control algorithm utilized locks, hence the relations used in a query must first be locked before any processing is carried out. Locking can be achieved in a distributed fashion by utilizing the transaction information broadcast in the Initial Broadcast Phase. If all relations are to be locked initially before the transaction is processed, this phase is completed at each related site when all needed local relations are not locked by other transactions. However, if the precedence relationship is resolved whenever conflicts are detected, as proposed in the concurrency control algorithm in Section IV, then this phase has to be repeated before each semijoin broadcast.

2) Local-Processing (LP) Phase: The LP phase may be started immediately at a related site as soon as the CC Phase is completed. Selections, projections, and local joins are performed in this phase. After local processing, the relations needed by the query in a related site are joined together into a single relation called the *site-relation*. An attribute in a site-relation is called a *site-attribute*. For clarity, the original relations and attributes are referred to as the *original-relations* and *original-attributes*. The site-relations and site-attributes are used in the discussion of the query processing strategy. The use of the original-attributes will be discussed after the various phases are presented.

Depending on the lock requests received from the bus, the results of the LP Phase for a transaction may have to be retracted if a conflicting lock request is received before the first semijoin broadcast for this transaction is sent. This problem can be avoided by broadcasting a message to inform all sites that the LP Phase for a transaction has started, so other sites will not send conflicting lock requests. The synchronization of the LP Phase will be discussed in the next section. The LP Phase for all related sites should be completed before the next phase can begin. This requires the identification of the slowest site and will be discussed in the next section.

3) Global Semijoin (GSJ) Phase: In this phase, semijoins on site-attributes are carried out. An initial site-attribute is broadcast, and site-relations at other related sites are semijoined with the broadcast site-attribute. The statistic of the site-attributes are then collected at each related site. Based on measures to be discussed in Section III-D, the site-attribute with the minimum *heuristic value*, called the *minimum attribute*, is selected as the next site-attribute to be broadcast. An algorithm to identify the minimum site-attribute is discussed in the next section. A semijoin may be blocked because one or more relations containing the attribute to be semijoined are locked by other transactions. To avoid the delay of waiting for these relations to be unlocked, the processing order of semijoins may be rearranged, so attributes that are not locked can be processed first. The process of selecting and broadcasting minimum attributes is repeated until all site-attributes are broadcast.

Semijoins can be improved by the use of complement attributes [30]. The *initial complement* denotes the complement of a site-attribute before the GSJ Phase. As global semijoins are carried out, the size of a site-attribute will decrease, while the size of its complement will increase. The size of the complement of an updated site-attribute will always be larger than that of its initial complement. Fig. 4 shows the relationships among the complement, the initial complement, and the site-attribute itself. When the cardinality of the updated site-attribute is smaller than that of its initial complement, a *normal broadcast* that broadcasts the updated site-attribute should be used; otherwise, a *complement broadcast* that broadcasts the initial complement should be used. A semijoin can be carried out with either the site-attribute or its complement.

Since the materialization is redundant, the information contained in one site-attribute may be a subset of the information contained in another site-attribute at a different site. For example, if $\boldsymbol{R}_1$ is stored in sites 1 and 2, then the information for the site-attributes at sites 1 and 2 will overlap. More than one copy of each original-relation may be joined together in this phase. The result of joining redundant copies of the same relation is equivalent to that of using a unique copy of the relation because the result of equijoining a relation to itself is equivalent to the relation itself, and join operations are commutable. However, redundant information should not be broadcast. The following definitions are defined for the method of eliminating redundancy.

Domain j of a query is defined as the union over all relations referenced by the query of all known possible values of attribute *j* in these relations before the query is processed. For example, if there are three relations, $\boldsymbol{R}_1, \boldsymbol{R}_2$, and $\boldsymbol{R}_3$, with attribute *a*, and $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ are referenced by the query, then domain *a* for this query is the union of all known possible values in attribute *a* of $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$.

The *component set* of site-attribute *j* in site-relation $\boldsymbol{R}_x$ is defined as the set of original-relations in site *x* that contain site-attribute *j*. For example, let $\boldsymbol{R}_1$ with attributes *a* and *b* ($R_1 \cdot a$, $R_1 \cdot b$) and $\boldsymbol{R}_2$ with attributes *b* and *c* ($R_2 \cdot b$, $R_2 \cdot c$) be stored in site *x*. After the LP Phase, a site-relation (R_x)

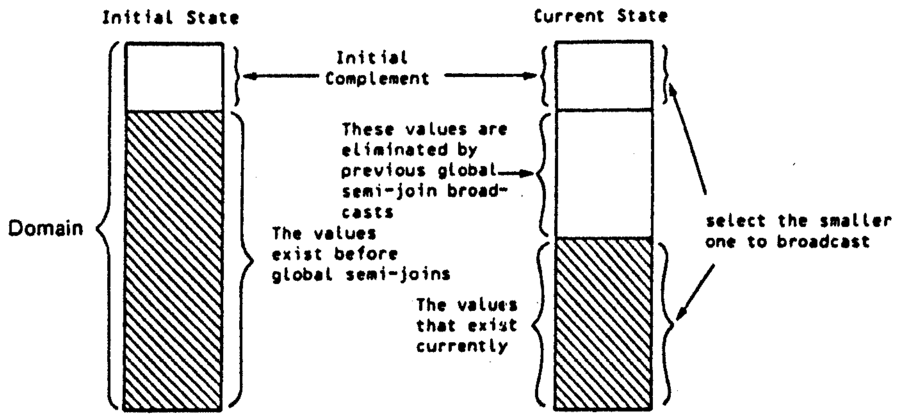

Initial State
Current State
Initial Complement
These values are eliminated by previous global semi-join broadcasts
Domain
The values exist before global semi-joins
The values that exist currently
select the smaller one to broadcast

Fig. 4. The set of values of an attribute, its complement, and the initial complement. (The initial state is defined at the time after the LP Phase and before the GSJ Phase. Since attribute values eliminated during the LP Phase at one site are unknown to other sites, the initial complement should include values in the domain that are eliminated during the LP Phase.)

with site-attributes $R_x \cdot a$, $R_x \cdot b$, and $R_x \cdot c$ is created by joining R_1 and R_2. The component sets of $R_x \cdot a$, $R_x \cdot b$, and $R_x \cdot c$ are $\{R_1\}$, $\{R_1, R_2\}$, and $\{R_2\}$, respectively.

The *broadcast component set of domain j* is defined as the union of the component sets of site-attributes in domain j that have been broadcast. A site-attribute j does not have to be broadcast if its component set is a subset of the broadcast component set of domain j, because the information contained in this site-attribute is already known to all sites. After a site-attribute has been broadcast, the corresponding broadcast component set is updated at all sites. For example, suppose R_1 is stored in sites x and y, R_2 is stored in site y, and R_3 is stored in site z. Assuming that these relations have one common site-attribute j, then site-attribute $R_x \cdot j$, with component set $\{R_1\}$, is constructed at x, site-attribute $R_y \cdot j$, with component set $\{R_1, R_2\}$, is constructed at y, and site-attribute $R_z \cdot j$, with component set $\{R_3\}$, is constructed at z. If $R_y \cdot j$ is broadcast first in the GSJ phase, then the broadcast component set of domain j is updated from empty to $\{R_1, R_2\}$. Since the component set of $R_x \cdot j$ is a subset of the broadcast component set of domain j, $R_x \cdot j$ does not have any reduction effect on $R_z \cdot j$ and does not have to be broadcast.

4) Relation-Transmission (RT) Phase: In this phase, the resulting fragments of the site-relations are broadcast sequentially to the *post processing sites*, where the complete join will be performed. The problem here is to determine the order of broadcasts. Since semijoins may not execute a join completely [2], an appropriate broadcast order may result in some reduction effects (join restrictions) on the unsent fragments. However, these effects are insignificant, especially when the number of semijoins in the GSJ Phase is large. Therefore, the site-relations may be broadcast in an arbitrary order. Of course, site-relations at the post processing sites do not have to be broadcast.

5) Post-Processing (PP) Phase: In this phase, the full join and subsequent operations are executed, and the results sent to the site from which the query originates. The process of identifying the post processing sites is different for retrievals and updates. For a retrieval, the set of post processing sites must be selected such that the total overhead of transmission in the RT Phase and transmission of results from the post processing sites to the originating site is minimum. Since the set of post processing sites must be identified before the RT Phase begins, and the difference in overheads of processing at different post processing sites is usually small, the originating site can be chosen as the post processing site. For updates, the results of the post processing sites must also be sent to all sites containing relations to be updated. By an argument similar to that of retrievals, the originating site is chosen as the post processing site. At the end of the RT Phase, the originating site will decide whether to broadcast the update set or the necessary relations stored at this site. Since all relations broadcast in the RT Phase are received by sites containing relations to be updated, the update set can be constructed if the necessary relations at the originating site are broadcast. Of course, consistency checks must be performed before the corresponding relations are updated. The query is completed after this phase.

The processing of query $Q1$ in Fig. 2 is illustrated in Fig. 5 $Q1$ is a relational query with the following predicate.

$$(R_1 \cdot a = R_3 \cdot a) \text{ AND } (R_1 \cdot b = R_2 \cdot b = R_4 \cdot b)$$
$$\text{AND } (R_2 \cdot c = R_3 \cdot c = R_4 \cdot c)$$

The distribution of the original-relations and the resulting site-relations are shown in Fig. 5(a). The various phases of query processing are shown in Fig. 5(b).

The size of a site-relation at a site may be much larger than the total size of the original-relations there. To reduce the cost of constructing the site relation, a site-attribute can be extracted from the original-attribute when needed. As a result, minor changes have to made in several phases. In the LP Phase, local semijoins, instead of full joins, are performed in each related site among the original-attributes. Since only equi-joins are considered, the set of distinct values in a site-attribute at a related site is the same as that in any original-attribute of the same domain there. Therefore, broadcasting a site-attribute $R_x \cdot j$ in the GSJ Phase is equivalent to broadcasting any origi-

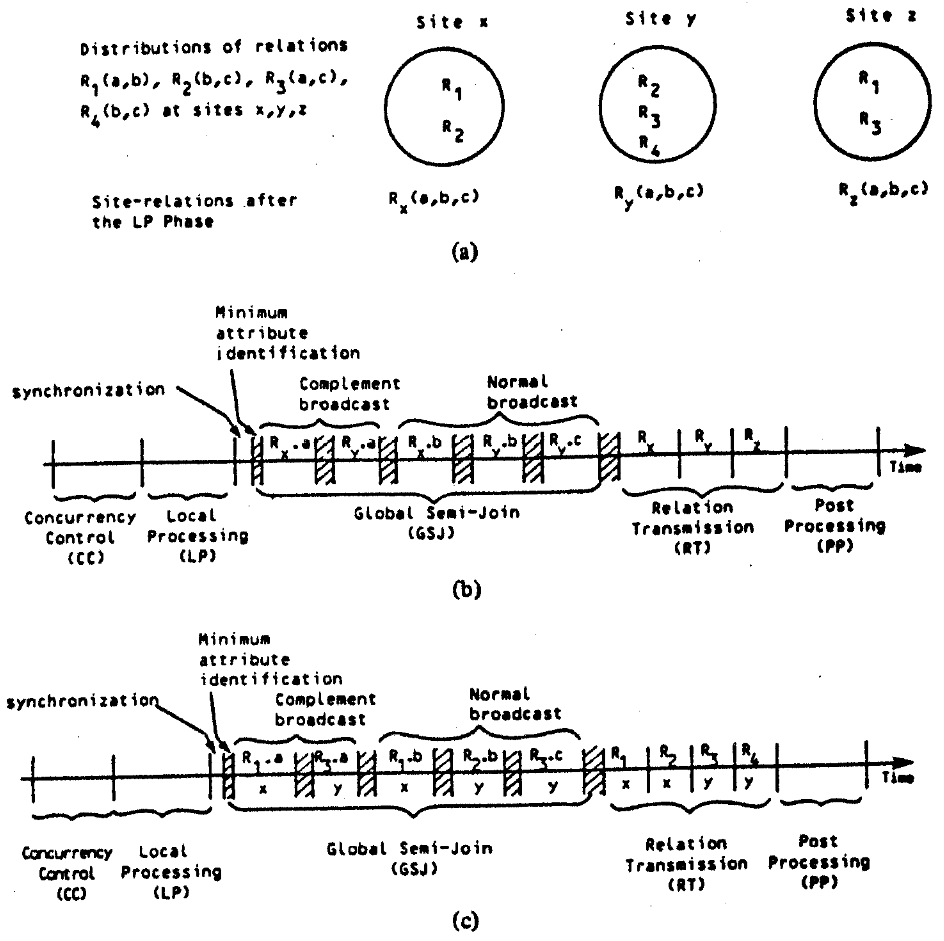


Fig. 5. (a) Distribution of relations and the result of the LP Phase. (b) An example of the schedule generated by the basic query processing strategy (the shaded area is the time for semijoins, synchronization, and identification of the minimum attribute; $R_x \cdot a$ is the site-attribute a at site x). (c) An example of the schedule generated by the improved query processing strategy (it is assumed that all required relations are locked in the CC Phase; the shaded area is the time for semijoins, synchronization, and identification of the minimum attribute; $R_k \cdot a$ is the original-attribute a of relation R_k; the site containing R_k is marked below $R_k \cdot a$ in the figure).

nal-attribute of domain j in site x. After an attribute is broadcast, all original-relations are semijoined with this attribute. In the RT Phase, fragments of the original-relations, instead of the site-relations, are transmitted. A unique copy of each relation must be selected for broadcast. Since the directory is redundant and the difference in sizes of the original-relations tends to be small after the GSJ Phase, a tie-breaking rule using the site number can be applied to identify the unique copy to broadcast. An example of the broadcast sequence using the original-attributes for the schedule in Fig. 5(b) is illustrated in Fig. 5(c).

C. Protocol for Extremum Identification

In the proposed query processing strategy, it is necessary to identify the completion of a phase and the minimum attribute. These two problems are equivalent to finding the site with the extremum value and can be solved by the protocol discussed below. Although the proposed protocol does not require additional hardware support, extremum-identification protocols with special hardware interface can also be used [18], [25]. Without loss of generality, only the problem of identifying the site with the maximum value is studied here. The algorithm is shown in Fig. 6. It is a distributed algorithm, executed at each site, that determines whether the local parameter y is the maximum. Each site contends for the bus and broadcasts its local value. It also listens to the bus for values broadcast by other sites. If the value received is better than y, then it drops out from further contention. This process continues until all sites have either broadcast or have been eliminated. The last broadcast value is then the maximum.

This process is complicated by the nondeterministic time to generate the local parameter at each site. After a site has broadcast its local parameter, it sets a timer to allow sites with larger parameters to broadcast their parameters and sites that have not finished evaluating their parameters to broadcast an "unfinished" message. If a timeout occurs before such a message is received, it will presume that the current maximum is the real maximum. Since the elapsed time before receiving such a message depends on the network load, the delivery of these messages should be given higher priority, especially for "unfinished" messages. To avoid the problem of flooding the network with unfinished messages, all sites that are still evalu-

```
process maximum (x, y);
{ /* This process executed at site x with value y identifies whether y is the maximum of all sites.
     current_max represents the broadcast maximum.
     next_x is the site that has been selected to broadcast next */

current_max := -∞;
compute next_x;   /* estimate the site that will broadcast next */
fork (generate_y);   /* create process to generate y */

while (true) do {
      wait (event);
      case (event) of {

         (y is generated): {
            if (x = next_x) then   /* broadcast y only if this site is next_x */
               signal network layer to contend for bus and to broadcast y;
            case (y) of {
              (y > current_max): current_max := y;
              (y ≤ current_max): return(failure) }

         (receive new_current_max): {
            current_max := max(new_current_max, current_max);
            next_x := x;   /* any site could broadcast next */
            case (y) of {
              (y is not available): {set timer; signal to contend for bus to broadcast 'unfinished'};
              (y > current_max): {set timer; signal to contend for bus to broadcast y};
              (y ≤ current_max): {
                 if (trying to broadcast y) then withdraw this broadcast;
                 return(failure) } } };

         (receive 'unfinished'): {
            unset timer;   /* there are unfinished sites */
            next_x := site sending the 'unfinished' message;
            if (trying to broadcast) then withdraw this broadcast };

         (get bus): {
            if (y is available) then {broadcast(y); set timer}
            else {broadcast('unfinished'); unset timer};

         (timeout): {
            if (trying to broadcast) then {
                 /* timeout occur, all unfinished sites stop any further broadcast */
                 withdraw this broadcast;
                 return(failure)}
            else return(success) }
         }
      }
}
```

Fig. 6. Protocol to identify the site with the maximum.

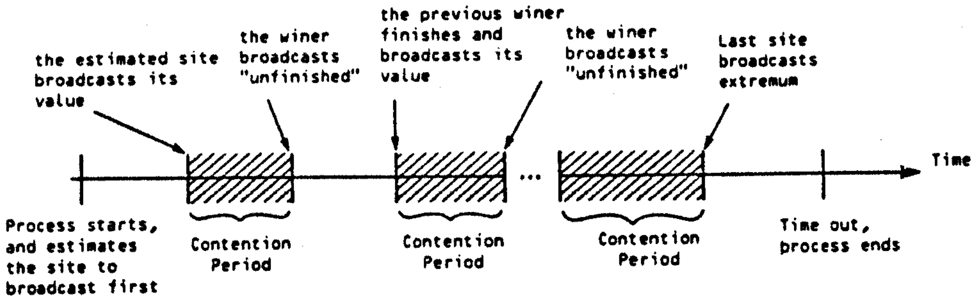

the estimated site broadcasts its value
the winer broadcasts "unfinished"
the previous winer finishes and broadcasts its value
the winer broadcasts "unfinished"
Last site broadcasts extremum
Time
Process starts, and estimates the site to broadcast first
Contention Period
Contention Period
Contention Period
Time out, process ends

Fig. 7. The sequence of events in identifying the extremum.

ating their parameters will refrain from sending further "unfinished" messages once the first "unfinished" message is received. The sequence of events is illustrated in Fig. 7. The shaded interval indicates a contention period during which sites with larger parameter or "unfinished" message contend for the bus. The contention period ends when an "unfinished" message is broadcast.

The performance of the proposed protocol can be enhanced by making a good initial estimate of the site containing the maximum. In the best case, only one broadcast and a timeout period are necessary. If the distribution of the parameters is known, then the maximum can be identified in a constant number of messages (on the average) independent of the number of sites [18], [25]. On the other hand, if the initial estimate is random, the average number of messages is $O(\log_2 N)$, where N is the total number of sites.

The above extremum-identification procedure can be applied to identify the completion of the LP Phase. Parameter y in Fig. 6 represents the local completion time, and the site with the latest completion time is sought. The time that each site spends between the Initial Broadcast and the end of the LP Phase is determined by the length of the CC and LP Phases. Since the time to resolve conflicts in each site is known globally (all remote actions are broadcast), and the processing times of selections, projections, and local semijoins are proportional to the cardinality of the relations, a good estimate of the slowest site can be made by all sites. All other sites must wait for the estimated slowest site to broadcast its completion time before determining the next "unfinished" site. To simplify the problem, the time of broadcast can be considered as the completion time of the slowest site, and only a "finished" message to indicate completion has to be broadcast. After the estimated slowest site has broadcast its "finished" message, the remaining unfinished sites contend for the bus to elect another site to broadcast the next "finished" message. If all sites have finished, the last site waits for a timeout period before asserting that the LP Phase has completed.

The procedure in Fig. 6 can also be applied directly to identify the completion time of each semijoin and the minimum attribute in the GSJ Phase. After one elected site has broadcast its heuristic value, finished sites with larger heuristic values are dropped from further contention, and unfinished sites contend to elect the next site to broadcast. If all sites have finished, sites with smaller heuristic values contend to elect the site with the minimum heuristic value. Since the estimation of the site containing the minimum attribute could be computationally expensive, it may be better to randomly elect one site as the site with the minimum attribute by the contention process. Another improvement is to allow each site to estimate its heuristic value before completion, and to drop out from contention if the estimated heuristic value is larger than the broadcast value. In this case, the next GSJ broadcast can be started without waiting for the slower sites with larger heuristic values to finish.

D. Computing the Heuristic Value

Good heuristic values, which identify the minimum attribute for broadcast in the GSJ Phase, are vital if the total amount of data transmissions is to be minimized. This heuristic function is affected by the following factors.

1) The size of an attribute is important because it affects the broadcast cost directly. Only the *effective size*, which is the product of the domain width and the larger of the current cardinality of the attribute and the cardinality of its initial complement, should be used.

2) Attributes with larger size-reduction effects on other attributes should be broadcast first because they affect the broadcast cost in successive semijoins.

3) The number of unbroadcast attributes on a common domain is another factor of consideration. When this number is large, the broadcast of any attribute in this domain will have a greater size-reduction effect on other relations.

It is very difficult to have a heuristic function that satisfies all the above rules. Depending on the application, the designer has to combine these parameters with appropriate weights to get a good heuristic function.

The selectivity of an attribute is usually used to estimate its size-reduction effect. *Selectivity* is defined as the ratio of the current cardinality of the attribute to the initial cardinality of the domain containing the attribute. When an attribute is semijoined with another attribute, its new size is computed as the product of its current size and the selectivity of the other attribute. This estimation method is not suitable in a broadcast network because the selectivity of a broadcast attribute may be used more than once in computing the size of other attributes. For example, suppose attributes $R_x \cdot j$, $R_y \cdot j$, $R_z \cdot j$, with sizes $s_{x,j}$, $s_{y,j}$, $s_{z,j}$, and selectivities $p_{x,j}, p_{y,j}, p_{z,j}$, are the attributes in domain j that are broadcast in sequence. After $R_x \cdot j$ is broadcast, the sizes of $R_y \cdot j$ and $R_z \cdot j$ are changed to $s_{y,j} \times p_{x,j}$, and $s_{z,j} \times p_{x,j}$, and their selectivities are changed to $p_{y,j} \times p_{x,j}$ and $p_{z,j} \times p_{x,j}$. Then $R_y \cdot j$ is broadcast next, the size of $R_z \cdot j$ is estimated to be $s_{z,j} \times p_{x,j} \times p_{x,j} \times p_{y,j}$, in which $p_{x,j}$ is used twice.

To overcome the above problem, the relative selectivity is used to measure the size-reduction effect of broadcast attributes. The *relative selectivity* of an attribute in domain j is the *domain* j, where current-domain j is domain j with the reduction effects taken into account. When an attribute in domain j with relative selectivity p is broadcast, the cardinalities of all attributes in domain j, as well as the cardinality of current-domain j, are reduced by a factor p. As a result, the relative selectivities of all attributes in domain j will not be changed, and the duplicate reduction effects using selectivities will not occur. In the last example, suppose $p'_{x,j}$, $p'_{y,j}$, and $p'_{z,j}$ represent the relative selectivities. Initially, $p'_{x,j} = p_{x,j}$, $p'_{y,j} = p_{y,j}$, and $p'_{z,j} = p_{z,j}$. After $R_x \cdot j$ is broadcast, the sizes of $R_y \cdot j$ and $R_z \cdot j$ are changed to $s_{y,j} \times p'_{x,j}$ and $s_{z,j} \times p'_{z,j}$, but then relative selectivities remain unchanged. Hence, when $R_y \cdot j$ is broadcast, the reduction effects on $R_z \cdot j$ is estimated to be $p'_{x,j} \times p'_{y,j}$.

Four different heuristic functions have been evaluated:

1) effective size of an attribute,

2) effective-size × selectivity (selectivity in the conventional sense),

3) relative selectivity,

4) effective-size × relative-selectivity.

These heuristic functions were evaluated by simulations on randomly generated relations. The number of relations and the size of each are restricted by the time it takes to find the optimal semijoin order by exhaustive enumeration. On the VAX 11/780 computer, it took about three minutes of CPU time to simulate a case of three relations with two common attributes each, and 24 hours of CPU time to simulate a case of three relations with three common attributes each. It was assumed that one copy of each relation existed in the database, and that each existed at a distinct site. The widths of the three attributes were six, eight, and ten, respectively. The width of tuples in each relation, including attributes not in the joining domains, was 30. The domain of each joining attribute was assumed to be integers between one and ten. The distinct values in an attribute were generated by first selecting a threshold, and then including a domain value in the attribute if a random number generated exceeded the threshold. After the attribute values were generated, a cross-product was formed for each relation to obtain the set of possible tuples. Tuples in this set were randomly selected for the relation by a process similar to that of selecting the attribute values. A number of database instances were obtained by using different thresholds.

The simulation results are shown in Table I (a) and (b). The ratios of the total communication costs of various heuristic schedules to that of the optimal schedule are compared. These results demonstrate that dynamic query processing, using relative selectivity as the heuristic function is very effective. The merits of the heuristic functions must be evaluated again when the relations are generated differently.

IV. Concurrency Control

A. Timestamping versus Locking

The design of the concurrency control algorithm depends on the requirements of the database. When the processing order of a set of transactions can be independent of their arrival order, the general condition to be enforced is *serializability* [20]. An interleaved execution of transactions at different sites is serial-

TABLE I
RELATIVE PERFORMANCE OF DIFFERENT HEURISTIC SCHEDULES AS COMPARED TO THE OPTIMAL SCHEDULE (PERFORMANCE OF THE OPTIMAL SCHEDULE IS ONE). (a) AVERAGE BEHAVIOR OF 64 CASES FOR THREE RELATIONS WITH TWO COMMON ATTRIBUTES EACH. (b) AVERAGE BEHAVIOR OF 17 CASES FOR THREE RELATIONS WITH THREE COMMON |ATTRIBUTES EACH.

(a)

	Effective-Size	Effective-Size × selectivity	Relative Selectivity	Effective-Size × Relative-Selectivity
Average	1.912	2.037	1.138	2.037
Stand. Dev.	0.338	0.388	0.080	0.388
Maximum	2.537	2.537	1.200	2.537
Minimum	1.099	1.099	1.022	1.099

(b)

	Effective-Size	Effective-Size × selectivity	Relative Selectivity	Effective-Size × Relative-Selectivity
Average	1.742	1.070	1.002	1.058
Stand. dev.	0.459	0.215	0.004	0.206
Maximum	2.200	1.876	1.017	1.878
Minimum	1.000	1.000	1.000	1.000

the transactions serially. The read (or write) set of a transaction is the set of data items that are read (or written) by the transaction. A *conflict* occurs between two transactions if the write set of one transaction intersects with the read set (r-w conflict) or write set (w-w conflict) of the other. A set of transactions can be executed in parallel and be serializable if the execution order of conflicting transactions are carefully arranged.

Most of the existing serializable concurrency control algorithms are based on either timestamps or locks [3]. In a *timestamp-ordering protocol*, a transaction is assigned a unique timestamp equal to the value of the local clock when it is initiated. In addition, each data item is also timestamped with its most recent read and write. Before a transaction is processed, its timestamp is compared with the timestamps of the accessed data. A transaction is rejected if a conflict is detected. When a transaction is rejected, it should be restarted with a new timestamp, and all intermediate results have to be discarded.

In a *locking protocol*, data items are locked before they are accessed. Locks must be issued in such a way that the transactions are serializable, and if deadlock occurs, recovery must be possible. Locks are released after the access is completed. A popular locking protocol is the two-phase protocol in which unlocks for a transaction are issued after all locks have been made. Lock based protocols avoid the expensive restarts of timestamp based protocols. However, they incur the following overheads when the network does not have the broadcast capability.

1) Messages for Requesting/Releasing Locks: For each data access, three sets of messages are needed: a set of messages for requesting locks on all accessed data, a set of acknowledgments in granting the locks, and a set of messages for releasing the locks.

2) Waiting Time for Requesting/Releasing Locks: The delay between requesting locks and granting locks is wasted when this transaction does not conflict with other transactions.

3) Extra Overhead for Deadlock Prevention, Detection, and Recovery: Since deadlocks may occur in lock-based protocols, additional overheads are incurred on deadlock detection and recovery. The degree of concurrency may be degraded when deadlock prevention is enforced.

4) Extra Waiting Time for Overlocking: A data item may be locked unnecessarily to minimize the possibility of deadlocks and to enforce serializability. Overlocking exists in protocols that request locks initially and in two-phase protocols.

The above overheads are significantly reduced in a broadcast network. Since activities in a broadcast network are known to all sites, each site can maintain a *Broadcast Transaction Table* that records the status of all transactions in the system and the status of referenced relations (locked or unlocked). All Broadcast Transaction Tables are always identical. When a transaction is processed, only the local Broadcast Transaction Table has to be checked to determine the serializability condition. A transaction will wait until serializability is satisfied. Moreover, explicit lock and unlock messages are not needed because a data item is locked when the corresponding transaction is broadcast, and unlocked when it is no longer needed by the transaction. Hence, the waiting times for requesting and releasing locks are saved. Overlocking is also reduced because an item is unlocked as soon as it is not used by the transaction. Further, deadlocks can be detected or prevented based on information in the Broadcast Transaction Table.

For these reasons, lock-based protocols should be used on a broadcast bus. An efficient protocol is proposed in the remainder of this section. The processing order of transactions is governed by their Initial Broadcast Times or the times that a conflict is found. It is assumed that the transaction processing model discussed in Sections I and III is used, that a relation is the basic unit of a data item, and that an access may be either a read or a write but not both. With a common bus and assuming that locks received are processed in real time before a lock request is broadcast, only lock requests that will not be rejected by other sites are broadcast. In this sense, the common bus acts as a gateway for concurrency control as in a centralized database.

B. Minimum Locking Concurrency Control Algorithm

In this section, we present an efficient algorithm that locks all relations for the minimum amount of time with respect to transactions that have higher precedence. The algorithm detects access conflicts between two transactions when they are broadcast; however, the order of processing is not imposed until the conflicting access of one of these transactions is made. After the order of processing is defined between TR_1 and TR_2 (say TR_1 precedes TR_2), then all accesses made by TR_1 must precede accesses of TR_2 to enforce serializability. It must be pointed out that minimum locking does not imply the maximum degree of concurrency.

The time for a transaction to read-lock a relation is the time that the relation is first read by the transaction. Consistency checks are performed before the relation is locked. For relations that are semijoined in the LP Phase, they should be locked before the LP Phase begins. For other relations that belong to the read set of the transaction but are not used in the LP Phase, they should be locked once one of their attributes is either broadcast of semijoined with a broadcast attribute. When the access to a relation is read only, a copy of the relation (with only the attributes to be accessed) is made in the working storage when it is read-locked, and the relation is unlocked immediately. If duplicate copies cannot be made due to storage limitation, then the relation must be read-locked until it is no longer read by the transaction. A relation that is updated must be write-locked at the beginning of the PP Phase by the first query in the transaction that updates the relation, and released at the end of the PP Phase by the last query in the transaction that updates the relation. For relations that are read and written in the same transaction, the read and write locks must be released together. The locks on the relations are kept in the Broadcast Transaction Table. The times to lock and unlock a relation are known to all sites without explicit messages because the directory and status information are fully distributed in the system.

The distributed concurrency control algorithm is shown in Fig. 8. The precedence relationships among transactions are stored in a precedence graph and is maintained by procedure *consistency* [Fig. 8(c)]. Procedure *precedencecheck* [Fig. 8(b)] checks whether two transactions conflict with each other.

Process *cc_minimumlock* [Fig. 8(a)] is triggered by the arrivals of lock requests, the successful contention in getting the bus, the termination of transactions, and the release of locks. Transactions originating from a site communicates with the local process *cc_minimumlock* there. Each transaction is a process, which can be in one of the following states: THINKING, SUSPENDED, PENDING, or ACTIVE. When a transaction is in the THINKING state, it does not generate any request to the database. After a transaction, say TR_i, makes a request, consistency checks are made on a temporary copy of the precedence graph to determine if any conflict exists between TR_i and other transactions in the precedence graph. If a conflict is found, then TR_i is SUSPENDED and processed later. When no conflict exists, TR_i is put into the PENDING state, so lock messages received from other sites can be processed. TR_i will be SUSPENDED if a conflict is found in this state; otherwise, it will signal the network layer to contend for the bus and will be put into the ACTIVE state. The ACTIVE state is terminated when the bus is granted to TR_i and the request is serviced. At this time, the original precedence graph is updated. If a lock request received conflicts with TR_i when it is ACTIVE, then TR_i is SUSPENDED and the request to contend for the bus is withdrawn.

Procedure *consistency* [Fig. 8(c)] maintains the precedence graph. It defines the processing order of two transactions if they have (r-w or w-w) conflict in accessing a common relation. Once this order is defined, all accesses with an r-w conflict on a common relation by a transaction of lower precedence must be initiated after the conflicting relation has been unlocked by transactions of higher precedence. For transactions with a w-w conflict, the transaction with a higher precedence can be waived because the effect on the database is equivalent to the update by the other transaction alone. To allow transactions to be processed efficiently, the precedence order should be defined as late as possible because the transactions are processed asynchronously, and the transaction to first make the conflicting access is unknown at the Initial Broadcast Time. If a conflict is detected when a transaction, say TR_i, is requesting an access, and if an edge does not exist between the two nodes representing TR_i and the conflicting transaction, say TR_j, in the precedence graph, an edge is added between the nodes. Transitive edges are also added to the precedence graph

```
process cc_minimumlock (BTT, PG);
/* This process serves the asynchronous arrivals of lock requests, unlock requests, and termination of
   transactions. The relations are locked for the minimum amount of time with respect to precedent
   transactions. It is assumed that a transaction can lock one relation in each request, and that only one
   request is processed at a time (due to the single bus). A transaction that has been initiated can be in
   one of the following states: PENDING: all lock requests received must be checked first to determine if
   they conflict with the current request; ACTIVE: the transaction is waiting for the availability of the
   bus; SUSPENDED: the transaction is not processed because there is another transaction that is
   ACTIVE or PENDING, or it conflicts with another transaction that has been broadcast from another
   site; and THINKING: transaction is not making any lock request. When a transaction is finished,
   either "waive" or "success" will be returned, and the transaction will be put in the THINKING state.
   Information about the active transactions and locked relations are available in BTT, the Broadcast
   Transaction Table. Information about the precedence relations of transactions are stored in PG, the
   Precedence Graph. */

{   while (true) do {
        wait (event);

         case (event) of {
           (arrival of lock request from TR_i): {
              if (there is a PENDING or ACTIVE request) then
                 set state of TR_i to SUSPENDED and suspend TR_i
              else {
                 PG' = PG;  /* make temporary copy of PG */
                 if (consistency(TR_i, PG') = success) then
                    if (new lock requests have been received from other sites) then
                       set state of TR_i to PENDING
                    else {
                       set state of TR_i to ACTIVE;
                       signal network layer to contend for bus }
                 else {
                    if (consistency(TR_i, PG') = waive) then { update BTT; return (waive) }
                    else set state of TR_i to SUSPENDED and suspend TR_i;
                    if (there are SUSPENDED requests) then wake up one SUSPENDED request }}}
           (receive lock request from TR_i from bus): { /* new lock is granted at another site */
              update BTT and PG;
              if (there is a PENDING or ACTIVE request TR_j) then {
                 update PG';
                 if (precedencecheck(TR_i, TR_j, PG') = reject or waive) then {
                    if (TR_j is ACTIVE) then
                       withdraw bus-contention request;
                    if (TR_j is to be rejected) then
                       set state of TR_j to SUSPENDED and suspend TR_j
                    else return (waive);
                    if (there are SUSPENDED requests) then wake up one SUSPENDED request } } }
           (a transaction is terminated at another site): update BTT, PG, and PG' (if necessary);
           (unlock is received from bus): update BTT;
           (bus is obtained): {
              broadcast lock request for ACTIVE transaction;
              PG = PG';    update BTT for locks obtained by the broadcast transaction;
              return (success);
              if (there are SUSPENDED requests) then wake up one SUSPENDED request } }
    }
}
```

(a)

```
procedure precedencecheck (TR_global, TR_local, PG);
/* This procedure checks the precedence graph PG to determine whether the lock requested by TR_local
   conflicts with TR_global. TR_global is the transaction that generates the new lock; TR_local is the local
   transaction that is in the PENDING or ACTIVE state. */

{  /* check the consistency for r-w conflicts */
   if (edge exists from N_global to N_local in PG and there is r-w or w-w conflict) then
      return (reject);
   else if (edge exists from N_local to N_global and there is a w-w conflict) then
      return (waive);
   else return (success)
}
```

(b)

```
procedure consistency (TR_i, PG);
/* TR_i is the requesting transaction. Transaction TR_i is requesting to read or write Relation R and is
   represented as a node N_i in PG. This procedure checks (and updates if TR_i is a new transaction) PG
   for conflicts. An edge will be added when one of the conflicting relations has been accessed.
   All transitive arcs exist in PG since the precedence relation is transitive */

{  if (N_i does not exist in PG) then add a node N_i to PG;

   /* check TR_i for r-w conflicts */
   for (all nodes N_j in PG ≠ N_i) do {
      if (there is no edge between N_i and N_j) then
         if (TR_i has r-w conflict with TR_j) then
            if (R has not been accessed by TR_j) then {
               add an edge from N_i to N_j;  add all necessary transitive arcs }
            else if (R is locked by TR_j) then {
               add an edge from N_j to N_i;  add all necessary transitive arcs; return (reject) }
            else { /* R is unlocked by TR_j */
               add an edge from N_j to N_i;  add all necessary transitive arcs };
      else if (there is an edge from N_j to N_i)
      and (r-w conflict is on R that has not been released by TR_j) then
            return (reject) }
   if (TR_i is a read request on R) then return (success);   /* all r-w conflicts have been checked */

   /* check TR_i for w-w conflicts if TR_i is making a write request on R */
   for (all nodes N_j in PG ≠ N_i) do {
      if (there is no edge between N_i and N_j) then
         if (TR_i has w-w conflict on R with TR_j) then {
            /* several alternatives exist here, the strategy used minimizes the number of updates */
            add an edge from N_i to N_j; add all necessary transitive arcs; return (waive) }
      else if (edge exists from TR_j to TR_i) and (R is write-locked by TR_j) then
         return (reject)   /* write on R for TR_i will be waived later if write has not started */
      else if (edge exists from TR_i to TR_j) and (TR_i has w-w conflict with TR_j on R)
      and (R is locked or has been unlocked by TR_j) then
         return (waive) };  /* R has been modified by another transaction of lower precedence */
   return (success)   /* all w-w conflicts have been checked */
}
```

(c)

Fig. 8. (a) Process for concurrency control with minimum locking time. (b) Procedure for checking precedence relation between two transactions. (c) Procedure for maintaining the precedence graph.

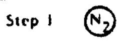
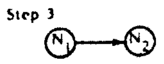
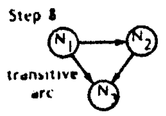
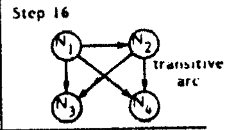
STEP	TR$_1$	TR$_2$	TR$_3$	TR$_4$	PRECEDENCE GRAPH
1		wlock B			
2		unlock B			
3	rlock A				
4		$\boxed{\text{wlock A}}$			
5	unlock A	delayed			
6		wlock A			
7		unlock A			
8			rlock B		
9			unlock B		
10	wlock C				
11			$\boxed{\text{wlock C}}$		
12	unlock C		delayed		
13			wlock C		
14		$\boxed{\text{wlock C}}$ waived			
15			unlock C		
16				wlock A	
17				unlock A	

Fig. 9. An example to illustrate the concurrent execution of four transactions as controlled by *cc_minimumlock* (the time that the transaction is initiated is the time for its first request; the four transactions are assumed to exist after the last step; rlock and wlock stand for read-lock and write-lock, respectively).

since the precedence relationship is transitive. The rules for determining whether a transition is accepted, rejected, or waived are specified in *consistency*.

An example of concurrent accesses of four transactions as controlled by the proposed algorithm is shown in Fig. 9. Boxed requests are not executed because they are either rejected or waived. In Step 4, when TR$_2$ requests a write-lock on A, there is an r-w conflict between TR$_1$ and TR$_2$, and TR$_2$ is delayed because an edge exist in the precedence graph from N_1 to N_2. In Step 6, the write request for TR$_2$ can proceed because A has been unlocked. In Step 8, N_3 is added to the graph. Since B was accessed by TR$_2$, an arc is included from N_2 to N_3, and a transitive arc from N_1 to N_3 is also added. TR$_3$ is allowed to proceed because B has been released by TR$_2$. In Step 14, the request to update C by TR$_2$ is waived because the update on C by TR$_3$ has been carried out and would have over-written the update by TR$_2$ (an edge exists from N_2 to N_3).

The serializability and deadlock-free properties of the proposed algorithm are proved in the following theorems.

Theorem 2: The sequence of requests as controlled by *cc_minimumlock* represents a serializable execution of the transactions.

Proof: Serializability for a read-only write-only model is satisfied when the requests are scheduled according to an acyclic precedence graph [20], [24]. To prove that the sequence of requests as scheduled by *cc_minimumlock* are serializable, it is necessary to show that the precedence graph as maintained by *consistency* is acyclic, and that *cc_minimumlock* follows this precedence graph in scheduling the requests. The precedence graph is acyclic because transitive arcs are always added whenever a new arc is added to the graph, and no arc will be added between two nodes if an arc already exists there. To show that the resulting precedence graph is followed by *cc_minimumlock* in scheduling all requests, it is seen that the precedence relationship of currently active transactions as defined by the precedence graph is always followed whenever a lock request is granted. It is noticed that an arc is added to the precedence graph whenever a conflicting access is first discovered between two transactions, and that this arc is maintained until one of these transactions is terminated. Hence, the resulting precedence graph obtained at the end of a sequence of requests can be assumed to exist before the requests are executed, and a linear order of execution can be enforced for all transactions, which results in a serializable execution. ∎

Theorem 3: The proposed protocol *cc_minimumlock* is deadlock-free.

Proof: Given a set of transactions TR$_1$, TR$_2$, $\cdots$, TR$_n$, the wait-for relationship in an execution controlled by *cc_minimumlock* can be represented in the following wait-for graph. For each transaction TR$_i$, there is a node N_i in the wait-for graph. A directional arc connects N_i to N_j if TR$_i$ waits for TR$_j$ on a conflicting access. Since only transactions of lower precedence must wait for transactions of higher precedence, and the precedence graph is acyclic, the wait-for graph is also acyclic, and the algorithm is deadlock free. ∎

The proposed concurrency control algorithm does not impose the precedence relationship between two transitions until the first conflict between them is detected. This allows shorter transactions to proceed first without waiting for the longer ones. However, the maintenance of the precedence graph at each site may pose a significant overhead there. In this case, a predefined precedence relationship may be used when the transactions are initiated. One convenient way is to define a higher precedence for a transaction with a smaller Initial Broadcast Time. The serializability and deadlock-free properties will still be satisfied.

C. Local Transactions

We have assumed that the status of all transactions is known to all sites in the proposed concurrency control algorithm. However, a local transaction that only accesses local relations may block other global transactions and is unknown to other sites. A transactions is local if all the relations to be read are available locally, and the relations to be written exist in a single copy and are available locally. To solve this problem, additional status messages can be broadcast during the processing of local transactions. This overhead may be large because a database with a good data distribution would have many local transactions. Another method is to allow the local transactions to be processed when the accessed relations are not locked based on information in the Broadcast Transaction Table. When a global transaction wishes to access a relation that is locked by a local transaction, it will be blocked until the relation is unlocked by the local transaction. To delay other global transactions, the site at which the local transaction originates will broadcast an "unfinished" message when other sites are trying to identify the completion of the LP Phase or the semijoins in the GSJ Phase. When the local transaction is finished, this site will process the pending global transactions and broadcast a "finished" message when completed.

V. Conclusions

In this paper, we have studied the design of a distributed database on a local computer system connected by a multiaccess network. Due to the broadcast capability of multiaccess networks, each update can be processed in a cost independent of the placement and number of copies. As a result, the file allocation problem is simplified, and polynomial-time algorithms for optimal placements have been obtained for some cases (although the problem is NP-hard in general). A transaction processing model that integrates consistency check and query scheduling has been proposed. In scheduling queries for distributed processing, redundant materialization, and semijoins with complement transmissions were adopted. A broadcast attribute can be semijoined with attributes at all sites simultaneously, hence reducing the amount of data transfer as compared to that of general networks. The broadcast network allows efficient distribution of status information and permits queries to be scheduled dynamically based on the current status of the database. Although dynamic query processing algorithms are suboptimal, they eliminate the errors incurred in estimating status information in static algorithms, and have been found to perform better than static algorithms. An extremum-identification algorithm was proposed to aid the dissemination of status information. Lastly, an efficient, serializable, and deadlock-free concurrency control algorithm was developed. The broadcast capability allows locks to be known to all sites simultaneously, and hence eliminates the use of explicit lock messages.

There remain problems to be studied in the future. These include the problem of recovery when the network or a site fails, the reliable broadcast of messages, the use of multiple broadcast busses or a bus with high bandwidth, the validity of the proposed algorithms when the local processing overhead cannot be neglected, and the necessary modification on the operating system to support real-time processing of information received from the bus.

REFERENCES

[1] P. M. G. Apers, A. R. Hevner, and S. B. Yao, "Optimization algorithms for distributed queries," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 57-68, Jan. 1983.

[2] P. A. Bernstein and D. M. Chiu, "Using semi-joins to solve relational queries," *J. ACM*, vol. 28, no. 1, pp. 25-40, Jan. 1981.

[3] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185-221, June 1981.

[4] 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 Trans. Database Syst.*, vol. 6, no. 4, pp. 602-625, Dec. 1981.

[5] S. Ceri, S. Navathe, and G. Wiederhold, "Distribution design of logical database schemas," *IEEE Trans. Software Eng.*, vol. SE-9, no. 4, pp. 487-504, July 1983.

[6] S. Ceri and G. Pelagatti, *Distributed Databases: Principles and Systems.* New York: McGraw-Hill, 1984.

[7] A. L. P. Chen and V. O. K. Li, "Improvement algorithms for semijoin query processing programs in distributed database systems," *IEEE Trans. Comput.*, vol. C-33, pp. 959-967, Nov. 1984.

[8] D. M. Chiu, P. A. Bernstein, and Y. C. Ho, "Optimizing chain queries in a distributed database system," *SIAM J. Comput.*, vol. 13, no. 1, pp. 116-134, Feb. 1984.

[9] W. W. Chu, "Multiple file allocation in a multiple computer system," *IEEE Trans. Comput.*, vol. C-18, pp. 885-889, Oct. 1969.

[10] W. W. Chu and P. Hurley, "Optimal query processing for distributed database systems," *IEEE Trans. Comput.*, vol. C-31, pp. 835-850, Sept. 1982.

[11] E. F. Codd, "A relation model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377-387, 1970.

[12] L. W. Dowdy and D. V. Foster, "Comparative models of the file assignment problem," *ACM Comput. Surveys*, vol. 14, no. 2, pp. 287-313, June 1982.

[13] K. P. Eswaran, "Placement of records in a file and file allocation in a computer network," in *Information Processing, 74, IFIPS.* Amsterdam, The Netherlands: North Holland, 1974.

Guide to the Theory of NP-completeness. San Francisco, CA: Freeman, 1979.

[15] A. M. Geoffrion and R. E. Marsten, "Integer programming: A framework and state-of-the-art survey," *Management Sci.*, vol. 18, no. 9, pp. 465-491, May 1972.

[16] M. G. Gouda and U. Dayal, "Optimal semi-join schedules for query processing in local distributed database systems," in *Proc. ACM SIGMOD Conf.*, pp. 164-175, May 1981.

[17] A. R. Hevner and S. B. Yao, "Query processing in distributed database systems," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 177-187, May 1979.

[18] J. Y. Juang and B. W. Wah, "Unified window protocols for contention resolution in local multiaccess networks," in *Proc. 1984 IEEE INFOCOM*, San Francisco, CA, Apr. 1984, pp. 97-104.

[19] R. Krishnamurthy and S. P. Morgan, "Distributed query optimization: An engineering approach," in *Proc. Int. Conf. Data Eng.*, Los Angeles, CA, Apr. 1984, pp. 220-227.

[20] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631-653, Oct. 1979.

[21] C. V. Ramamoorthy and B. W. Wah, "The isomorphism of simple file allocation," *IEEE Trans. Comput.*, vol. C-32, pp. 221-232, Mar. 1983.

[22] G. M. Sacco, "Distributed query evaluation in local area networks," in *Proc. Int. Conf. Data Eng.*, Los Angeles, CA, Apr. 1984, pp. 510-516.

[23] A. S. Tanenbaum, *Computer Networks.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[24] J. D. Ullman, *Principles of Database Systems.* Rockville, MD: Computer Science Press, 1982.

[25] B. W. Wah and J. Y. Juang, "An efficient protocol for load balancing on CSMA/CD networks," in *Proc. 8th Conf. Local Comput. Networks*, Minneapolis, MN, Oct. 1983, pp. 55-61.

[26] B. W. Wah, "File placement on distributed computer systems," *IEEE Computer*, vol. 17, no. 1, pp. 23-32, Jan. 1984.

[27] B. W. Wah and Y. N. Lien, "The file-assignment and query-processing problems in local multiaccess networks," in *Proc. Int. Conf. Data Eng.*, Los Angeles, CA, Apr. 1984, pp. 228-235.

[28] E. Wong, "Retrieving dispersed data from SDD-1: A system for distributed databases," in *Proc. 2nd Berkeley Workshop Distributed Data Management and Computer Networks*, May 1977, pp. 217-235.

[29] S. B. Yao, "Optimization of query evaluation algorithms," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 133-155, June 1979.

[30] C. T. Yu, C. C. Chang, and Y. C. Chang, "Two surprising results in processing simple queries in distributed databases," in *Proc. COMPSAC 82*, Nov. 82, pp. 377-384.

[31] C. T. Yu and C. C. Chang, "On the design of a query processing strategy in a distributed database environment," in *Proc. SIGMOD Conf.*, vol. 13, no. 4, 1983, pp. 30-39.

Benjamin W. Wah (S'74-M'79) received the B.S. and M.S. degrees in electrical engineering and computer science from Columbus University, New York, NY, in 1974 and 1975, respectively, and the M.S. degree in computer science and the Ph.D. degree in engineering, both from the University of California, Berkeley, in 1976 and 1979, respectively.

Currently, he is an Associate Professor in the School of Electrical Engineering, Purdue University, West Lafayette, IN. His current research interests include parallel computer architecture, distributed databases, and theory of algorithms.

Dr. Wah has been a Distinguished Visitor of the IEEE Computer Society since 1983.

Yao-Nan Lien received the B.S. degree from National Cheng Kung University, Taiwan, China, in 1979, and the M.S. degree from Purdue University, West Lafayette, IN, in 1981, both in electrical engineering.

Currently, he is working toward the Ph.D. degree in electrical engineering at Purdue University. His research interests include distributed databases, operating systems, computer