Seoul Journal of Business
Volume 6, Number 1/2 (December 2000)

A Decision Support Tool for distributed Database Design

Sangkyu Rho*
College of Business Administration
Seoul National University

Salvatore T. March
Carlson School of Management
University of Minnesota

Abstract

The efficiency and effectiveness of a distributed database depend primarily on solving two interrelated design problems: *data allocation*, specifying what data to replicate and where to store it, and *operating strategies*, specifying where and how retrieval and update processes are performed. We develop a distributed database design approach that comprehensively addresses these problems, explicitly modeling their interdependencies for both retrieval and update processing. We extend earlier distributed database design models to include join order and data reduction by semijoin, in addition to data replication, copy identification, and join node selection. We demonstrate that join ordering and data reduction by semijoin are important distributed database design decisions that must be included in a distributed database design algorithm if it is to determine an overall optimal distributed database design.

I. Introduction

Geographically distributed organizations are faced with the

* This research was partially supported by Institute of management Research, College of Business Administration, Seoul National University. (Corresponding author)

challenge of developing information systems that efficiently support local operations and enable information sharing across the organization [Brancheau, et al., 1996]. Distributed database systems supply the underlying technology for such systems, providing users with access to databases that are maintained at different locations [Ricciuti, 1993; Richter, 1994; The, 1994]. They can yield significant cost and performance advantages over centralized systems for geographically distributed organizations [Ozsu and Valduriez, 1991a; 1991b].

Given a computer network consisting of *nodes* containing computers with processing and storage capabilities that are connected by *links* with data transmission speeds and capacities, judicious placement of data and processing capabilities can result in efficient and responsive systems. However, inappropriate data replication and placement or inappropriate processing of that data can result in high cost and poor system performance [Ceri, et al., 1987].

There are two aspects to distributed database design, *data allocation* and *operating strategies*. Data allocation includes the determination of units of data to allocate, termed *fragments*, and the placement of copies of those units on nodes in the network. To enhance retrieval efficiency, the same fragment can be redundantly allocated to multiple nodes, resulting in multiple copies of the same data. Such redundancy results in more complex update processes and increased data maintenance costs.

Operating strategies include *operation allocation*, or query optimization, and *concurrency control* strategies. Operation allocation defines where, how, and in what order retrieval and processing operations are performed [Yu and Chang, 1984]. The concurrency control strategy is responsible for ensuring that update operations are performed correctly and consistently, a particularly challenging task when there are multiple copies of the data [Bernstein and Goodman, 1981].

Retrieval operations must be performed at nodes that contain the required data. Processing operations can be performed at any node; however, if the data are not located at the processing node, they must first be sent there over the communication network. The order in which operations are performed can have a significant impact on performance. To reduce the amount of

processing required, selection and projection operations are always performed before join operations. The order in which join operations are performed and the use of data reduction strategies, or *semijoins*, can also have significant impact on performance [Yoo and Lafortune, 1989]. Update operations must eventually be done at all nodes containing a copy of the affected data.

Prior work has produced distributed database design approaches that optimize either overall cost or average response time [Apers, 1988; Blankinship et al., 1997; Cornell and Yu, 1989; Ram and Narasimhan, 1990; 1994; March and Rho, 1995; Rho and March, 1995]. These vary in the design decisions considered and the optimization procedures used. The most comprehensive work includes the generation and allocation of fragment replicas, the selection of retrieval and join nodes for retrieval queries, and the effects of concurrency control mechanisms for update queries, optimizing performance measures that include both local computer resources and network resources. None have included join order or data reduction strategies.

In this work we address the effects of join order and data reduction strategies on the overall task of distributed database design. Both have been shown to significantly affect performance in the context of query optimization [Mishra and Eich, 1992; Galindo-Legaria and Rosenthal, 1997; Yu and Chang, 1984]. Neither has been considered in distributed database design approaches. Optimizing join order and utilizing semijoins can significantly reduce the amount of processing that must be done and the amount of data that must be transmitted when performing distributed joins operations. Failure to consider these at design time overestimates the cost of join operations involving distributed data. This can result in the selection of more centralized designs and loss of the benefits afforded by data distribution.

If a distributed database design approach fails to consider join order or the utilization of semijoins, the cost of this query could be significantly overestimated. As a result, the cost of the above data allocation could be overestimated and it may be rejected, even if it is, in fact, the overall optimal one for this system.

We extend earlier distributed database design approaches to

include join order and data reduction by semijoins as well as data replication, copy identification, and join node selection. No current distributed database design approach includes all of these components. Our cost model can be used to evaluate operating cost or response time [Rho and March, 1995]. In this paper we focus on operating cost minimization. A response time minimization model that includes parallelism is under development [Johansson, 1999]. We utilize a genetic algorithm-based solution procedure to select data and operation allocations that minimize overall system operating cost within node, network, and query response time constraints. We have applied this procedure to a set of example problems. These demonstrate that operating cost can be significantly reduced when join order and data reduction by semijoin are considered.

The remainder of the paper is organized as follows. In the next section we briefly overview distributed database design concepts focusing on the effects of join order and data reduction strategies. In the next section we present our cost model and solution algorithm. Finally, we discuss the effects of data replication, join node selection, and data reduction strategies on the overall operating costs.

II. Distributed Database Design

In a distributed database system, data from a single conceptual database are maintained at various nodes in a computer network. The process of allocating data to nodes is termed distribution design or data allocation [Ceri et al., 1987, Ozsu and Valduriez, 1991a]. Given a data allocation, user retrieval and update queries must be processed. Queries arise at some node and may update or retrieve data stored at any node. The process of determining how, when, and where queries are processed is termed query optimization or operation allocation. A concurrency control mechanism specifies update-processing constraints.

Typically the data allocation and the concurrency control mechanism are determined at design time and change infrequently, if at all. Although there are research efforts in data migration strategies [Gavish and Sheng 1990], this aspect of

distributed system operation is beyond the scope of this paper. Operation allocation is typically done by a query optimizer within the distributed database management system either at compile time [Lohman et al., 1985] or at run time [Epstein et al., 1978]. We argue that it is important to generate an efficient operation allocation for each known query at design time. This enables designers to estimate the system load and to pre-compile query execution strategies. It also provides the necessary estimates of system load to determine efficient data allocations. Globally optimized query processing strategies may, in fact, be more efficient than one-at-a-time query optimization.

For illustrative purposes, consider a bank having four locations, headquarters and three regional offices. Suppose further that each location has a computer system (node) in a fully connected network. Each computer is described by its CPU and disk capacities and their unit costs. Each link in the network is described by its speed, capacity, and unit transfer costs. Suppose that the database schema has three tables, Customer, Account, and Transaction as in Figure 1. Each customer has some number of accounts against which deposit and withdrawal transactions are made. Each customer has a preferred regional office at which the customer does most of his/her banking, typically the office at which the accounts were opened. Of course, each regional office must be able to process transactions for any customer. Regional offices and headquarters require access to data about various customers, accounts, and transactions.

Figure 2 shows an example set of retrieval and update queries. Each is expected to be executed from each location with a specified selection criteria and frequency. For example, Retrieval Query R1 could be executed from headquarters once per day, selecting region 1 accounts. It could be executed once per hour from region 2 selecting region 2 accounts, and so forth. A distributed database system should allocate data and operations for efficient execution of known queries. Based on the retrieval queries R1, R2, and R3, for example, each relation in Figure 1 could be horizontally partitioned into three fragments, each containing the instances for one region. Each fragment could be allocated to each node at which the data are requested. This design enables efficient processing of retrieval queries but has

Customer (10,000 instances, 960,000 characters)

c-id	Text	5
c-name	Text	20
ssn	Text	9
c-address	Text	30
c-city	Text	20
c-state	Text	2
c-zip	Text	10

Account (15,000 instances, 1,350,000 characters)

acc-no	Text	8
c-id	Text	5
br-id	Text	5
a-type	Text	2
a-status	Text	2
s-balance	Numeric	15.2
s-date	Date	8
c-balance	Numeric	15.2
period-interest	Numeric	15.2
ytd-interest	Numeric	15.2

Transaction (3,000,000 instances, 23,400,000 characters)

t-id	Text	10
acc-no	Text	8
loc-id	Text	5
t-date	Date	8
t-time	Time	8
t-amount	Numeric	15.2
t-type	Text	2
t-status	Text	2
t-ref	Text	20

Figure 1. Tables for an Example Distributed Database System

significant data redundancy, possibly resulting in poor update query performance. Its overall performance depends on the frequency with which retrieval and update queries are executed and the response time requirements for each query.

Operation allocation, or distributed query processing, involves three phases [Yu and Chang, 1984]: copy identification, reduction, and assembly. *Copy identification* is required if more than one copy of a needed fragment exist. If so, the copy to use for the query is determined in this phase. The copy identification

a. Retrieval Queries

R1. Customer Statements

SELECT	c-id, c-name, c-address, c-city, c-state, c-zip, acc-no, s-balance, c-balance, period-interest, ytd-interest, t-id, t-type, t-amount
FROM	Customer, Account, Transaction
WHERE	Customer.c-id = Account.c-id
AND	Account.acc-no = Transaction.acc-no
AND	Account.br-id = [region]

R2. Balance Inquiry

SELECT	c-id, c-name, acc-no, c-balance
FROM	Customer, Account
WHERE	Customer.c-id = Account.c-id
AND	acc-no = [specified]

R3. Branch Status Report

SELECT	br-id, acc-no, c-balance
FROM	Account
WHERE	br-id = [region]

b. Update Queries

U1. Adjust Account balance

UPDATE	Account
SET	c-balance = [new balance]
WHERE	acc-no = [specified]

U2. Maintain Customer Data

UPDATE	Customer
SET	c-address = [specified], c-city = [specified], c-state = [specified], c-zip = [specified]
WHERE	c-id = [specified]

U3. Record Transaction

INSERT INTO	Transaction
VALUES	('t-id', ..., 't-ref')

Figure 2. Retrieval and Update Queries for an Example Database System

phase is also termed *materialization* because the data required must be "materialized," or retrieved from a specific node.

Reduction applies only to join queries when the fragments to be joined are stored at different nodes. In it, *semijoins* [Bernstein and Chiu, 1981] are used to reduce the amount of data that must be transferred to accomplish join operations. To join two fragments stored at different nodes, the required data from one

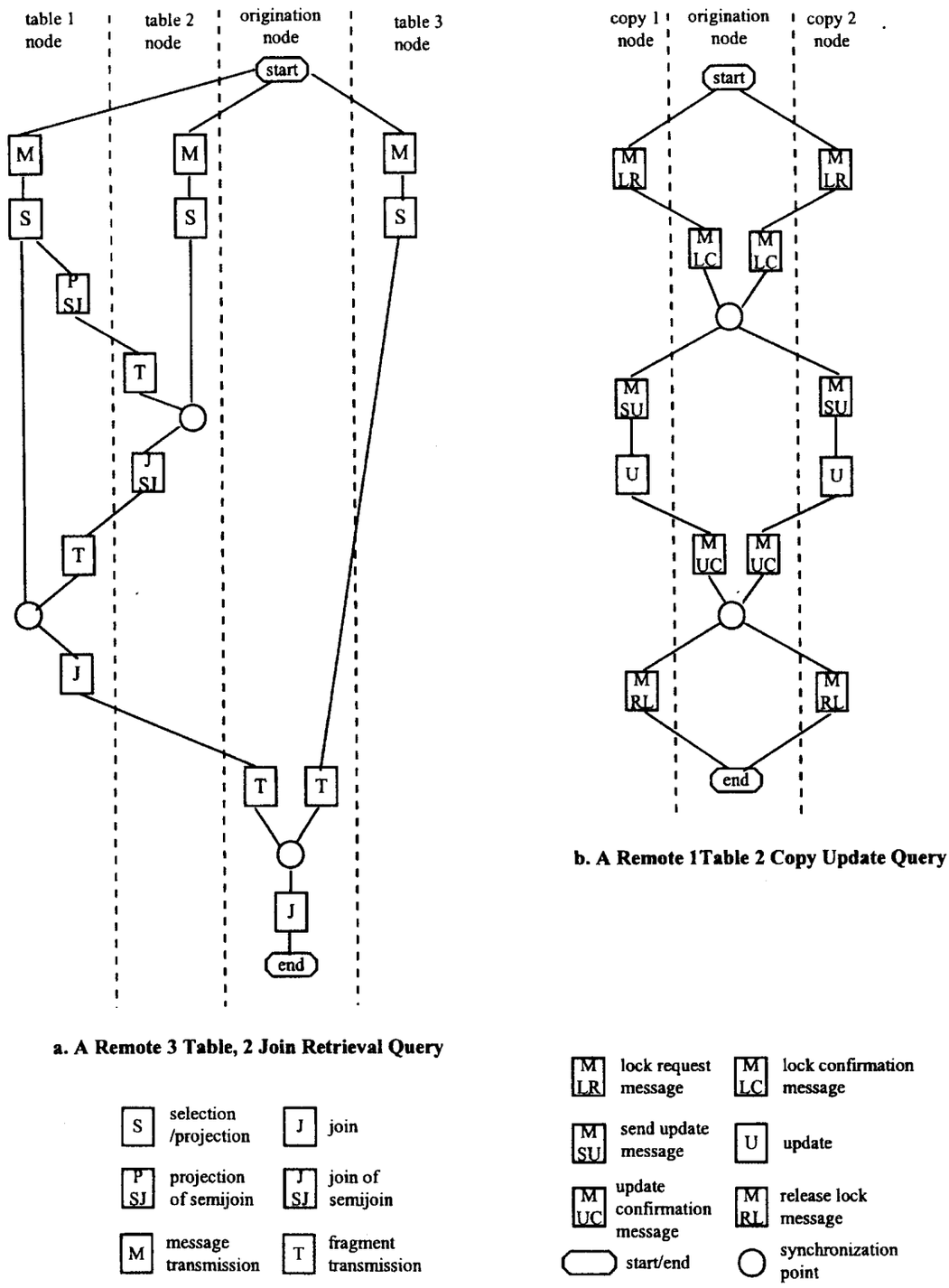
of the fragments must be transmitted to the node at which the other is stored, or the required data from both must be transmitted to a third node. If there are rows in one fragment without corresponding rows in the other fragment, data can be transmitted unnecessarily.

As discussed above, a semijoin can reduce the amount of data transmitted by identifying rows that have matching join values. It does as follows. One fragment is selected as the *reducer* and the other as the *reducee*. The unique join attribute values are projected from the reducer and transmitted to the node containing the reducee. A row in the reducee is selected if its join attribute matches one of the transmitted join values, i.e., a join is performed between the unique join attributes of the reducer and the reducee. The selected rows of the reducee are transmitted to the reducer node where the join is performed. A semijoin is effective, or *beneficial*, if its cost is less than the cost of sending the entire reducee fragment to the reducer node and performing the join there. Determining when semijoins are beneficial is a complex task, particularly when there are multiple, possibly cyclic, joins in the same query task [Yoo and Lafortune, 1989].

In *assembly* data are sent to the result node (if they are not already there) and final processing is performed (e.g., sorting and aggregations). Much of the research in distributed query optimization assumes that all reduced fragments are sent to the result node where all joins are performed. In this research the solution algorithm determines the nodes at which joins are performed and the join order.

These three phases correspond to query steps [Cornell and Yu, 1989] or operations, some of which can be processed in parallel and some of which must be processed sequentially [Rho, 1995]. As illustrated in Figure 3, each query has a start and an end (designated by ovals), a set of operations (designated by rectangles), and a set of synchronization points (designated by circles). At a synchronization point, all previous operations must be finished before subsequent operations can begin.

Retrieval queries require up to six types of operations: message, selection/projection (or restrict), join, projection of semijoin attribute values, join of semijoin, and data transmission [Rho, 1995]. These operations are used during



a. A Remote 3 Table, 2 Join Retrieval Query

b. A Remote 1Table 2 Copy Update Query

Figure 3. Retrieval and Update Query Processing Models

distributed query processing as follows. During the copy identification phase, messages are sent from the query origination node to the nodes from which data are retrieved. During the reduction phase selection/projection operations and semijoins, if any, are performed at these nodes. During the

assembly phase intermediate files are transmitted to join nodes where joins are actually performed. If not already there, results are transmitted to the query destination node.

Figure 3.a shows a possible execution plan for a three table, two-join retrieval query, such as R1 in Figure 2. In this example, each fragment (table) is retrieved from a different node, each of which is remote from the query origination node, and the query destination node is the same as the query origination node. It is executed as follows. First, messages are sent to each node from which a table is retrieved. Upon receiving these messages, the appropriate selection/projection operations are performed at each node. A semijoin is used for tables 1 and 2 as follows. The join attribute is projected from table 1 and transmitted to table 2's node where the reduction is performed. The reduced table 2 is transmitted to table 1's node where the join is performed. The join result is transmitted to the query destination (origination) node, where it is joined with table 3, which was transmitted there after appropriate selection and projection operations were performed at table 3's node. Join operations cannot begin until the needed data are available at the join node. Thus joins that require data from different nodes have a synchronization point prior to the join.

Assuming a 2PL concurrency control strategy, update queries also require six operations: lock request message, lock confirmation message, update message, local update, update confirmation message, and release lock message. Figure 3.b shows the execution plan for a remote update query where two copies of the affected table are allocated to different nodes. Synchronization points are required before updates are performed and before lock release messages are sent.

Many possible execution plans exist for each retrieval query depending on the data allocation design. Update execution plans are essentially fixed by the concurrency control strategy. A distributed database design algorithm must determine an efficient, if not optimal, data allocation for all fragments and an efficient, if not optimal, execution plan for each query. These must conform to capacity and query response time constraints.

III. A Cost Model for Data and Operation Allocation

Given a set of fragments (tables) to be maintained in a given network and a profile of retrieval and update queries that specify a set of min-term fragments [Apers, 1988], our approach performs the following tasks:

(1) allocate fragment replicas to nodes (data allocation),

(2) for each retrieval query:

- allocate query steps to nodes, identifying the appropriate fragment copies to use for retrieval queries (copy identification),
- identify beneficial semijoins for all join steps (reduction), and
- determine join order for join queries involving more than one join and a node at which each join is performed (assembly)

(3) for each update query, determine the update cost for the specified data allocation,

to minimize total operating cost within specified network and response time constraints.

In this section we summarize our operating cost model. Problem definition and solution components and cost equations are summarized in Appendix 1. Detailed equations are presented in [Rho, 1995]. In the following section we present a generic algorithm to select efficient solution components based on this cost model.

A Total Operating Cost Model

The allocation of costs to various operations in distributed systems is a difficult problem. It depends on such factors as hardware utilization, the actual variable costs of operation such as electricity and personnel, and the recovery of investment. Minimizing cost essentially results in a weighted minimization of required system resources, an important consideration in establishing and conforming a distributed information system budgetary requirements. Presumably, minimizing the computing resources required by this distributed database makes those resources available for other applications.

Our performance model is designed to minimize total

operating cost including communication, disk *I/O*, *CPU* processing, and storage. Simply stated, its objective is:

$$\text{Min Cost} = \sum_k f(k) \sum_m (COM(k, m) + IO(k, m) + CPU(k, m)) + \sum_t STO(t)$$

Where $f(k)$ is the frequency of execution of query k per unit time, $COM(k, m)$, $IO(k, m)$, and $CPU(k, m)$ are the respective costs of communication, disk *I/O* and *CPU* processing time for step m of query k, and $STO(t)$ is the cost of storage at node t per unit time. Thus the objective is to minimize the cost of each query step, times the frequency of its execution, plus the cost of data storage at each node.

Define $copy(i, t)$ as a 0-1 decision variable representing the allocation of fragment i to node t. That is, $copy(i, t)$ is 1 if fragment i is allocated to node t. It is 0 otherwise. Given a data allocation, storage costs for node t are straightforward. They are given by:

$$STO(t) = (\sum_i copy(i, t)\ L_i)\ s_t$$

where L_i is the length of fragment i, and s_t is the unit storage cost per unit time at node t. L_i and st are problem parameters.

Communication, disk *I/O* and *CPU* processing costs are represented as per unit costs multiplied by the number of units used. Let c_{tp} be the communication cost per character from node t to node p and let $H(k, m, t, p)$ be the amount of communication on the link connecting these nodes due to step m of query k. Then the overall communication cost for step m of query k is given by,

$$\text{COM}(k, m) = \sum_t \sum_{p \neq t} H(k, m, t, p)\ c_{tp}.$$

Similarly, let d_t be the cost per disk *I/O* at node t and let $O(k, m, t)$ be the disk *I/O* load at that node due to step m of query k. Then the overall disk *I/O* cost for step m of query is given by,

$$IO(k, m) = \sum_t O(k, m, t)\ d_t.$$

Finally, let pt be the *CPU* processing cost per unit and let $U(k, m, t)$ be the number of *CPU* processing units expended at node t for local processing and communication for step m of query k. Then the total *CPU* processing cost for step m of query k is given by,

$$CPU(k, m) = \sum_t U(k, m, t)\ p_t$$

Expressions for $H(k, m, t, p)$, $O(k, m, t)$ and $U(k, m, t)$ depend on the decisions made for the other steps in the query. Representing them analytically is extremely difficult [Rho, 1995]. The number of decision variables and constraints needed to do so explodes combinatorically, as discussed below.

Each step in a retrieval query requires a set of fragments and a set of operations needed to restrict and combine them. Message and selection/projection steps require only one fragment. Join steps require two fragments. First, consider message or selection/projection steps. Let $a(k, m)$ be the fragment required by step m, of query k. Define $op\_a(k, m, t)$ as a 0-1 decision variable having a value of 1 if a copy of $a(k, m)$ at node t is used for this query step. The cost of this query step is calculated as follows.

Messages must be transmitted from node t to node $orig(k)$, the query origination node, if they are different. These messages result in communication as well as local *CPU* and *I/O* processing costs at $orig(k)$ and at t. Local selection/projection costs are always incurred at node t. The amount of communication on the link connecting t and $orig(k)$ due to step m of query k is given by:

$$H(k, m, t, orig(k)) = 0 \qquad \text{if } t = orig(k)$$
$$H(k, m, t, orig(k)) = L^M * op\_a(k, m, t) \qquad \text{otherwise,}$$

where L^M is the length of a message. $O(k, m, t)$ and $U(k, m, t)$ are similarly calculated (see Appendix A). An $op\_a(k, m, t)$ decision variable is needed for each node and each message and selection/projection step in each query. Hence, the number of

decision variables for message and selection/projection is on the order of

Nodes * (message steps + selection/projection steps),

where Nodes is the number of nodes in the network.

For join steps, a second fragment, $b(k, m)$, must be included in the query step specification. The join operation can be allocated to any node in the network. Define $node(k, m, t)$ as a 0-1 decision variable having a value of 1 if join step m of query k is allocated to node t. As above, $op\_a(k, m, t)$ is used to specify the copy to use for $a(k, m)$; $op\_b(k, m, t)$ is used to specify the copy to use for $b(k, m)$. Messages must be sent if either fragment is retrieved from a node other than $orig(k)$ or if the join is not performed at $orig(k)$. Similarly, data transmission costs are incurred if either or both fragments are located at nodes different from that specified by $node(k, m, t)$. If a semijoin is performed, one fragment must be assigned the role of reducer and the other the role of reducee. Define $red\_a(k, m)$ as a 0-1 decision variable having a value of 1 if $a(k, m)$ reduces $b(k, m)$ and a value of 0 if it does not. Similarly define $red\_b(k, m)$. At most, one fragment can be the reducee. If neither is specified as the reducee, then the join is processed without reduction by semijoin.

If a query involves only one join, then the cost of each possible strategy could be calculated and multiplied by the appropriate combination of decision variables, e.g., the cost of executing step m of query k by reducing $a(k, m)$ from node t_1 by $b(k, m)$ from node t_2 at node t_3 must be multiplied by $red\_a(k, m) * op\_a(k, m, t_1) * op\_b(k, m, t_2) * node(k, m, t_3)$, itself a challenging non linear representation. However, if the query has two or more joins, then the cost of any join step also depends on the join order and on how the other join steps were processed. Hence a decision variable would need to be generated for each possible node for $a(k, m)$ and $b(k, m)$ and for each possible semijoin strategy for each possible join order for each join in the query.

To alleviate these problems, we have taken a genetic algorithm approach. Genetic algorithms work by generating, evaluating, and selecting solutions represented by a gene structure. It is not necessary to represent a solution space using decision variables

as in traditional optimization methods, thus eliminating the explosion of decision variables discussed above. Our gene structure represents solution components including the allocation of fragments and operations to nodes, the ordering of join operations, and alternative semijoin strategies. Solutions are generated and evaluated by the genetic algorithm descriptively rather than parametrically as in traditional optimization techniques. Our gene structure and genetic algorithm are presented in the next section.

System Constraints

To be feasible a data and operation allocation must satisfy certain intrinsic and capacity constraints. Furthermore, it may be desirable to place additional constraints on query response time, particularly in a cost minimization formulation. We enforce two types of intrinsic constraints: (1) all fragments must be allocated to at least one node and (2) each query step must be allocated to some node. We also enforce resource capacity constraints on each communication link and on disk *I/O*, *CPU*, and storage space on each node.

The intrinsic constraints are specified as follows. Each fragment must be assigned to at least one node (t represents nodes, i represents fragments):

$$\sum_t copy(i, t) \geqq 1 \qquad \text{for all } i.$$

Each query step must be assigned to a fragment copy (t represents nodes, k represents queries, m represents query steps):

$$\sum_t op\_a(k, m, t) = 1 \qquad \text{for all } k \text{ and } m.$$

$$\sum_t op\_b(k, m, t) = 1 \qquad \text{for all } k \text{ and } m.$$

Each join step must be assigned to a node:

$$\sum_t node(k, m, t) = 1 \qquad \text{for all } k \text{ and } m.$$

Furthermore, each message and selection/projection step must be allocated to a node at which a copy of the needed data exists, hence, the additional constraints:

$op\_a(k, m, t) \leqq copy(a(k, m), t)$ for all k, m and t.
$op\_b(k, m, t) \leqq copy(b(k, m), t)$ for all k, m and t.

At most one fragment in a join query step can be the reducee:

$red\_a(k, m) + red\_b(k, m) \leqq 1$ for all k and m.

Note that if $red\_a(k, m)$ and $red\_b(k, m)$ are both 0 then the join proceeds without reduction by semijoin.

Resource constraints are specified as follows. Communication link capacity:

$TL(t, p) \leq UL(t, p)$ for each link (t, p), $t = 1, 2, \ldots,$ number of nodes; $p = 1, 2, \ldots,$ number of nodes; and $t \neq p$.

Disk I/O capacity:

$TIO(t) \leq UIO(t)$ for each node t, $t = 1, 2, \ldots,$ number of nodes.

CPU capacity:

$TCPU(t) \leq UCPU(t)$ for each node t, $t = 1, 2, \ldots,$ number of nodes.

Storage capacity:

$G(t) \leq US(t)$ for each node t, $t = 1, 2, \ldots,$ number of nodes.

A response time constraint for query k can be specified as:

$R_{COM}(k) + R_{IO}(k) + R_{CPU}(k) \leq$ Required Response Time(k).

This represents a constraint on the sum of the response times

for each query step. Since it may be possible to process certain query steps in parallel, it is an upper bound on the response time of the query. Equations for the response time components $R_{COM}(k)$, $R_{IO}(k)$, and $R_{CPU}(k)$ are presented elsewhere [Rho and March, 1995]. The effects of parallelism on distributed database design are currently under investigation [Johansson, 1999].

The genetic algorithm enforces constraints by ascribing a large cost to any solution that violates any constraint. This can be easily done since the genetic algorithm generates and evaluates complete designs.

A Genetic Algorithm Solution Procedure

Adequate representation of the solution space and tractability of the solution approach are significant problems in distributed database design [Dowdy and Foster, 1988; Blankenship, et al., 1997]. To accurately reflect the interdependencies between data and operation allocation and to model the effects of semijoins and join order on query processing performance in a traditional optimization model, a large number of decision variables would need to be generated. Furthermore, problem and solution parameters interact in subtle and complex ways resulting in nonlinear and discontinuous objective functions and constraints (e.g., Temporary fragment sizes ($L_{a(k,m)}$ or $L_{b(k,m)}$) depend on the join order (See Rho and March [1997] for detail).). Therefore, it is impractical, if not impossible, to develop solution procedures based on traditional algorithms such as branch and bound.

To address these problems, we use a genetic algorithm-based solution procedure [Goldberg, 1989; Davis, 1991]. A genetic algorithm was chosen for several reasons. First, genetic algorithms work by generating and evaluating complete solutions. Hence, it is not necessary to represent a solution space or the interdependencies among problem and solution components using decision variables as in traditional optimization methods. Second, genetic algorithms are robust in that they work well even in discontinuous, multimodal, noisy search spaces. Genetic algorithm-based solution methods can easily incorporate very complex and nonlinear cost models such as ours. Third, genetic algorithms result not only in a "best" solution, but also in a pool of good solutions. The set of

solutions in the final pool provides significant intuition into the effects of design alternatives. For example, if all solutions in the final pool store a given file at a particular node, the designer would be reasonably confident that it is important to store that file at that node.

Our distributed database design algorithm contains a genetic algorithm within a genetic algorithm. Its basic structure is adapted from [March and Rho, 1995] and summarized in Appendix 2. As in that work, the outer genetic algorithm addresses data allocation while the inner genetic algorithm addresses operation allocation. We augment the gene structure and offspring generation algorithms to include join order and semijoin strategies in addition to copy identification and join node selection. These will be discussed after a brief overview of the genetic algorithm itself.

The outer algorithm begins by randomly generating a pool of feasible data allocations. The pool is "seeded" with heuristically generated solutions, including complete replication of all fragments at all nodes and the allocation generated by the "most beneficial sites" heuristic [Teorey, 1990]. For each data allocation in the pool, the inner genetic algorithm is used to determine a good, if not optimal, operation allocation for that data allocation. The inner genetic algorithm begins by generating a pool of feasible operation allocations including join order and data reduction strategies for each query for the given data allocation. It then iterates through generations, choosing operation allocations to be parents and combining them to produce children operation allocations. The best operation allocations are maintained in the pool at each generation to retain a fixed poolsize. After the specified number of iterations, the operation allocation with the best performance is selected, yielding a complete distributed database design including an efficient execution plan for each query.

After an efficient operation allocation has been produced for each data allocation in the initial outer algorithm pool, the outer algorithm similarly iterates through generations, choosing data allocations to be parents and combining them to produce children data allocations. The inner algorithm is executed for each child data allocation generated by the outer algorithm, yielding a complete design for it. After the specified number of

iterations, the complete solution with the best performance is selected and the algorithm terminates. In this way, only feasible data allocations are considered in the operation allocation algorithm, and the final solution is the best from among an evolving pool of good solutions.

Using outer and inner genetic algorithms can make it easier to handle the dependency between data allocation and operation allocation than using a single genetic algorithm representing both data and operation allocation. As discussed above, the feasibility of an operation allocation is dependent on the data allocation — each retrieval operation must be allocated to a node containing the required data. Update operations must be applied to all copies. It is very difficult to enforce these types of constraints in a single generic algorithm. Furthermore, such a nested approach allows us to easily incorporate different operation allocation models. Such flexibility is desirable in a distributed database design approach since different distributed database management systems utilize different query optimizers.

The genetic algorithm is written in C++ and runs in a UNIX environment. Its run time depends on problem size and on algorithm parameters such as the poolsize and number of iterations for each algorithm.

In the rest of this section we briefly describe the gene structure by which solutions are represented and the offspring generation processes used to search the solution space. Details of the algorithm are presented in Rho [1995]. Specifically we define a two-tiered gene structure. The first tier represents the data allocation. It corresponds directly to the decision variables represented by *copy*(*i*, *t*). The second tier consists of four parts, each representing one of the four types of decisions in our operation allocation model: (1) copy identification, (2) beneficial semijoin identification, (3) join order, and (4) join node selection. March and Rho [1995] propose a similar gene structure, however, they include only parts (1) and (4) since they do not consider semijoins or join order. Figure 4 shows the complete representation of a complete distributed database design solution for the sample problem. Each part of the representation is discussed below.

The data allocation is represented by sets of n bits, one set for each fragment, where n is the number of nodes in the network.

a. Data Allocation Gene Representation for the Outer Genetic Algorithm

Fragment	Fragment Allocation
Customer 1	1110
Customer 2	1010
Customer 3	1001
Account 1	1110
Account 2	0010
Account 3	0001
Transaction 1	0100
Transaction 2	0010
Transaction 3	0001

b. Operation Allocation Gene Representation for the Inner Genetic Algorithm

Query	Origination Node	Copy Id.	Semi-join	Join Order	Join Node
R1.1	HQ	0 0 1	00 10	2 1	0 1
R1.1	Region 1	1 1 1	00 00	1 2	1 1
R1.2	Region 2	0 0 1	00 10	2 1	2 1
R1.3	Region 3	0 1 1	01 00	1 2	1 3
R2.1	HQ	0 0	00		0
R2.1	Region 1	1 1	00		1
R2.1	Region 2	2 2	00		2
R2.2	HQ	0 2	10		0
R2.2	Region 1	2 2	00		2
R2.2	Region 2	2 2	00		2
R2.3	HQ	0 3	01		0
R2.3	Region 3	3 3	00		3
R3.1	HQ	0			
R3.1	Region 1	1			
R3.2	HQ	0			
R3.2	Region 2	2			
R3.3	Region 3	3			

Figure 4. An Example Solution Representation for the Genetic Algorithm

A bit has a value of 1 if the corresponding file fragment is allocated to the corresponding node. It has a value of 0 otherwise. Thus, each bit corresponds to *copy*(*i*, *t*). The example data allocation solution shown in Figure 4.a (1110 1010 1001 1110 0010 0001 0100 0010 0001) stores Customer 1 at Headquarters, Region 1, and Region 2; Customer 2 at Headquarters and Region 2; Customer 3 at Headquarters and Region 3; Account 1 Headquarters, Region 1, and Region 2; Account 2 only at Region 2; Account 3 only at Region 3; Transaction 1 only at Region 1; Transaction 2 only at Region 2; and Transaction 3 only at Region 3.

The operation allocation for each query is represented by four sets of vectors corresponding to the four columns, Copy Id, Semijoin, Join Order and Join Node illustrated in Figure 4.b. Each row in that figure contains the solution for a single variation of one of the retrieval queries in Figure 2. For example, Retrieval Query R1 in Figure 2 requires data from three tables, Customer, Account, and Transaction. Since there are three regions, it has three variations based on the selection condition, Account.br-id = [region]. These are designated R1.i in Figure 4.b, where i represents the Account.br-id selection condition. For example, R1.1 has Account.br-id = 1. Thus it requires the fragments, Customer 1, Account 1, and Transaction 1.

The copy identification vector has a position for each fragment referenced by a query. Each position holds the node from which the corresponding fragment is accessed. For example, the copy identification vector for R1.1 has three entries, one for each fragment. The selected copy identification for this query, when it originates at Headquarters, is the vector (0 0 1), as illustrated in the Copy Id column of the corresponding row in Figure 4.b. This specifies the use of Customer 1 and Account 1 from Region 0 (Headquarters) and Transaction 1 from Region 1. The selected copy identification for this query when it originates at Region 1, (1 1 1), uses all three tables from Region 1. When executed from Region 2, it uses Customer 1 and Account 1 from Headquarters and Transaction 1 from Region 1 (0 0 1). Copy identification decisions for the remaining queries are similarly represented.

A pair of bits represents each semijoin decision, one bit for each fragment in the join. The first bit represents *a*(*k*, *m*), the second represents *b*(*k*, *m*). The bit corresponding to the reducer

fragment is set to 1. Hence, the bit pair 10 represents $a(k, m)$ reduces $b(k, m)$ and the pair 01 represents $b(k, m)$ reduces $a(k, m)$. The bit pair 00 represents the decision not to use semijoins. The bit pair 11 is not legal. For example, Query R.1 (Figure 2) specifies two joins, (Customer join Account) and (Account join Transaction), extracted from the join specification Customer join Account join Transaction. Hence it needs two bit pairs to represent its semijoin strategy. The selected semijoin strategy for this query, selecting accounts in Region 1, R1.1, and originating at Headquarters is the vector of bit pairs (00 10), as illustrated in the Semijoin column of the corresponding row in Figure 4.b. This specifies the use of the semijoin Account 1 reduces Transaction 1. A semijoin is not used for the other join (it is performed at the query origination node). Semijoins are not used for this query when executed from Region 1 since all data needed are located at that region. Hence, its semijoin strategy is represented as (00 00).

Join order decisions are represented as a list of joins, where the sequence indicates the order in which joins are performed. The join order decision for query R1.1 originating at Headquarters is the list (1 2) shown in the Join Order Column of Figure 4.b. It specifies that the join between Customer 1 and Account 1, the first join listed in the query definition, is performed first and the join with Transaction 1, the second join listed in the query.

Join node decisions are represented by a vector with a position for each join in the query. Each position corresponds to a join step of the query and contains the node at which the join is performed. The join node decision for query R1.1 originating at Headquarters, the vector (0 0) in the Join Node column of Figure 4.b, specifies that both joins are performed at Headquarters.

Our genetic algorithm generates new solutions via standard genetic operators, constrained to ensure feasibility. Uniform crossover [Davis, 1991; Syswerda, 1989] and mutation are used for data allocation, beneficial semijoin identification, and join node selection. In uniform crossover, child genes are randomly selected from each parent. For example, consider the following data allocation solutions for the example Customer, Account, Transaction database:

1110 1010 1001 1110 0010 0001 0100 0010 0001
1110 1100 1000 0001 1000 1110 1100 0010 1100

Both solutions have the fragment Customer 1 stored at nodes 1, 2 and 3. Neither have it stored at node 4. In uniform crossover, all of children solutions from these parents will also have this fragment stored at nodes 1, 2 and 3. None will have it at node 4. Similarly, both parents have the fragment Customer 2 stored at node 1, but only the second parent has that fragment stored at node 2 and only the first parent has it stored at node 3. Neither have it stored at node 4. Again, in uniform crossover, all their children would have that fragment at node 1. They would have a .5 probability of having it at nodes 2 and 3. None would have it at node 4.

Uniform crossover is not viable for join order as it is very likely to generate children representing infeasible solutions. Therefore, we employ a *uniform* order crossover operator [Davis, 1991] for join order. In a uniform order crossover operator, gene positions for which a child will inherit values from the first parent are randomly determined. Values for the rest of the gene positions are determined based on the gene value order in the second parent, thus child solutions are always feasible. To illustrate how a uniform order crossover operator works, consider the following two join order solutions for a four join query:

2 1 3 4
1 3 4 2

In the first solution, join 2 is done first followed by joins 1, 3, and 4. Recall that the join number is specified in the statement of the query; joins can be performed in any sequence. In the second solution, join 1 is done first followed by joins 3, 4, and 2. When join order is generated for a child solution, gene positions from the first parent are randomly determined, i.e., the probability that a position is selected from the first parent is .5. Suppose that the second and fourth gene positions are selected from the first parent. We then have the following partial solution: - 1 - 4. Joins 2 and 3 are unspecified. In the second parent join 3 precedes join 2, thus the child join order would be 3 1 *2* 4. A second, complementary child could be generated with the

opposite selections. It would inherit 2 - 3 - from the first parent. Thus it would be 2 *1* 3 4.

Standard mutation operators frequently generate infeasible solutions for this type of representation since a standard mutation operator changes one gene in the solution. Thus *inversion* is used instead of mutation to incorporate randomness. Inversion generates a new solution by reversing the gene order of an existing solution. Under inversion two cut points are chosen at random and a child is produced by switching the end points of the middle segment. To illustrate how an inversion operator works, consider the first join order representation above, 2 1 3 4. Suppose that the point between the first and second genes and that between the third and last genes are chosen as cut points. The order of the two joins between the cut points is reversed from 1, 3 to 3, 1, thus the child join order would be 2 3 1 4.

To get a sense of how solution is represented and how its cost is calculated, consider again a three-table, two-join retrieval query, similar to R1, specified as,

```
SELECT    Customer.c-id, c-name, c-address, c-city, c-
          state, c-zip, Account.acc-no, s-balance, c-
          balance, period-interest, ytd-interest, t-id, t-
          type, amount, t-date
FROM      Customer, Account, Transaction
WHERE     Customer.c-id = Account.c-id
AND       Account.acc-no = Transaction.acc-no
AND       c-city = 'Minneapolis'
AND       t-date ≧ '12/1/99' AND t-date ≧ '12/31/99';
```

One possible query execution plan for such a query is illustrated in Figure 3a. Designating the query origination node as node 0 and the other nodes for the tables used, the gene structure for that solution is,

Copy Id	Semi-join	Join Order	Join Node
1 2 3	10 00	1 2	1 0

Where Customer is table 1, retrieved from node 1, Account is table 2, retrieved from node 2, and Transaction is table 3,

retrieved from node 3. The join of Customer and Account is performed first, at node 1, using a semijoin with Customer as the reducer and Account as the reducee. The join with Transaction is performed second at node 0, the query origination and result node.

Consider the cost components of this query execution plan. First, messages must be sent from node 0 to nodes 1, 2 and 3 requesting the needed data (query steps 1, 2 and 3). As described in Appendix 1, the communication cost of these message steps is simply $L^M * (c_{01} + c_{02} + c_{03})$ where L^M is the size of a message and c_{tp} is the unit cost of communication from node t to node p. The *CPU* costs of these messages is $(3 * S_0) * p_0 + R_1 * p_1 + R_2 * p_2 + R_3 * p_3$ where S_t is the *CPU* time required to send a message from node t, R_t is the *CPU* time required to receive a message at node t and p_t is the cost of *CPU* time at node t.

Next the selection/projection operations and the join operations must be performed. Selection and projection operations reduce the size of the target tables by applying the specified query selection and projection criteria to the appropriate tables. Join operations combine the reduced tables, perhaps using semijoins to further reduce the size of one of the tables.

The above query, has three selection/projection operations corresponding to query steps 4, 5, and 6. They are defined by the following subqueries:

```
(4) SELECT   c-id, c-name, c-address, c-city, c-state, c-zip,
    FROM     Customer
    WHERE    c-city = 'Minneapolis';

(5) SELECT   c-id, acc-no, s-balance, c-balance, period-
             interest, ytd-interest
    FROM     Account

(6) SELECT   acc-no, t-id, t-type, amount, t-date
    FROM     Transaction
    WHERE    t-date ≧ '12/1/99' AND t-date ≦ '12/31/99';
```

Each reduces the size of the table to be used in further operations. In this example, only relevant Customer and

Transaction rows are selected (c-city = 'Minneapolis' and t-date ≧ '12/1/99' AND t-date ≦ '12/31/99', respectively). All rows are selected from the Account table. Only relevant attributes are projected from each table.

The cost of selection/projection steps depends on the local database designs. If indexes exist on the selection criteria (c-city in the Customer table and t-date in the Transaction table) these can be used to minimize the local processing costs. Otherwise, the tables must be scanned. The *IO* costs for these steps are designated, $D_{k41} * d_1$, $D_{k52} * d_2$, $D_{k63} * d_3$, where k is the query designator, assumed to be the current query, D_{kmt} is the number of *IO* operations required to perform the select/project operation for step m of query k at node t and d_t is the cost per *IO* operation at node t. *CPU* costs are similarly calculated. Again, both depend on the local database designs, particularly indexing schemes and local query optimization strategies. Details of their calculations for various database design options are discussed in [Rho, 1995].

Finally, there are two join operations, Customer Join Account corresponding to query step 7 and Account join Transaction corresponding to query step 8. Since Customer is used in a semijoin to reduce Account, the unique c-id values from the selected Customer rows, i.e., those in Minneapolis must be projected from that table. This can be done in the selection / projection step. Call the result table, RedCustIDs (Reduced Customer Identifiers). For the join operation to proceed, this result table must be sent to node 2 at a cost of Size(RedCustIDs) * c_{12}, where Size(T) is the number characters in the table T. There it is used to select rows from the Account table that have a c-id value equal to one of those sent. That is, it executes the join query,

```
SELECT    Account.c-id, acc-no, s-balance, c-balance,
          period-interest, ytd-interest
FROM      RedCustIDs, Account
WHERE     RedCustIDs.c-id = Account.c-id;
```

Call the result of this join query, RedAccount (Reduced Account). The cost of producing this result depends on the local database design. Call it, CostRedAccount. This table is sent back

to node 1 at a cost of Size(RedAccount) * c_{21}. There it is joined with the Customer table as follows,

```
SELECT    Customer.c-id, c-name, c-address, c-city, c-state,
          c-zip, acc-no, s-balance, c-balance, period-
          interest, ytd-interest
FROM      Customer, RedAccount
WHERE     Customer.c-id = RedAccount.c-id
```

Call the result of this query JoinCustAcc (Joined Customer and Account). Again, the cost of producing it depends on the local database design. Call it CostJoinCustAcc. This table is sent to node 0 at a cost of Size(JoinCustAcc) * c_{10}. There it is joined with ResTrans (Restricted Transaction), the result of the selection/projection operation performed on the Transaction table in query step 6. ResTrans contains the columns acc-no, t-id, t-type, amount, t-date from the Transaction table whose t-date attribute is between '12/1/99' and '12/31/99' inclusive. The cost to produce it was discussed above. It was sent to node 0 at a cost of Size(RedTrans) * c_{30}.

Hence, the overall cost of this query is given by the sum of communication and local processing costs for messages, selection/projection operations, and join operations. The expressions used to calculate these costs depend on the order in which the operations are performed, the size of the intermediate results (which depends partially on the order in which operations are performed and the data reduction by semijoin strategy), the local database designs, and network and node costing parameters.

IV. An Evaluation of Solution Components

In this section we discuss the effects of data replication, join order, and data reduction by semijoin on distributed database design solutions. The sample problem used in this section has four relations: Salesperson, Customer, Order, and Product; and ten types of retrieval queries and five types of update queries executed with varying frequencies and selection criteria at different nodes.[1] We used the problem parameters in Table 1. We

Table 1. Cost Structure

Cost Component	Capacity	Cost
Link	5 Kbytes/sec	$2.00/Mbytes
Disk IO	400 IOs/sec	$2.50/M IOs
CPU	20 MIPS	$0.00005/MIPS
Storage	1 Gbytes	$10.00/Mbytes/month

conclude that each can have significant effects on the overall operating cost of a distributed database system. However, their effects are interdependent and different strategies are appropriate under different conditions. There are no simple rules of thumb. Hence, it is important for a distributed database design approach to consider their joint effects if it is to produce efficient solutions.

In a series of experiments discussed in detail elsewhere [Rho and March, 1995], we observe that replication alone is extremely effective for retrieval intensive environments when the proportion of rows required from any table in a query is "high." This corresponds to an operational reporting system where management requires detailed reports. In this environment (Figure 5.a) replication alone improved performance by nearly 60 percent over the base case (single copy, fixed join order, no data reduction). Join order and data reduction by semijoin yielded virtually no incremental performance improvement over replication alone. When replication was not considered, join order selection and data reduction by semijoin, in combination, yielded only a 33 percent performance improvement over the base case. Clearly replication dominates in such an operating environment.

This is reasonable since replication gains its efficiencies by storing copies of data wherever they are used. It thus reduces or even eliminates communication for retrieval processing, but increases update and storage costs. In a retrieval intensive environment, update costs are minimal and data storage costs are typically dominated by retrieval costs. Data reduction by semijoin only marginally improves performance since with high

1) Although it would be difficult to know all the queries in advance, we argue that a small number of known queries account for most of the query processing requirements in practice.

a. Retrieval Intensive Environment with High Proportion Selected

	Without Replication		With Replication	
	Cost ($)	Improvement over Base Case	Cost ($)	Improvement over Base Case
Base Case	11155	0.0%	4549	59.2%
Join Order and Semijoins	7498	32.8%	4508	59.6%

b. Update Intensive Environment with Low Proportion Selected

	Without Replication		With Replication	
	Cost ($)	Improvement over Base Case	Cost ($)	Improvement over Base Case
Base Case	4834	0.0%	4245	12.2%
Join Order and Semijoins	2840	41.3%	2836	41.3%

c. Mixed Retrieval and Update Environment with High Proportion Selected

	Without Replication		With Replication	
	Cost ($)	Improvement over Base Case	Cost ($)	Improvement over Base Case
Base Case	9650	0.0%	6838	29.1%
Join Order and Semijoins	7321	24.1%	6274	35.0%

d. Mixed Retrieval and Update Environment with Low Proportion Selected

	Without Replication		With Replication	
	Cost ($)	Improvement over Base Case	Cost ($)	Improvement over Base Case
Base Case	7269	0.0%	4549	17.1%
Join Order and Semijoins	3676	49.4%	4508	49.9%

Figure 5. Relative Performance Improvements of Replication and Sophisticated Operation Allocation Strategies

proportions of tables selected, there is little to reduce. Join order can still be important, but only to reduce local processing costs. In the extreme case when there are no updates and data storage is relatively inexpensive, self-contained nodes, i.e., nodes that contain a copy of all data needed at that node, are, in fact, optimal and distribution of operation allocation is irrelevant.

The opposite is true in update intensive environments when the proportion of rows required from any table in a query is "low." This corresponds to a transaction system with exception reporting only (very small subset retrieval). In this environment (Figure 5.b) join order selection and data reduction by semijoin without replication improved performance by 41 percent over the base case (single copy, fixed join order, no data reduction). Replication yielded virtually no incremental performance improvement over join order selection and data reduction by semijoin alone. When join order selection and data reduction by semijoin were not considered, replication yielded only a 12 percent performance improvement over the base case. Clearly join order selection and data reduction by semijoin dominate in such an operating environment.

Again, this is reasonable since join order selection and data reduction by semijoin can reduce data transmission requirements for both retrievals and updates, while replication reduces data transmission requirements for retrievals but increases it, and local processing costs, for updates.

Mixed retrieval and update environments show similar results. Replication is more effective when retrieval queries require a high proportion of rows (Figure 5.c). Join order and data reduction by semijoin are more effective when retrieval queries require a low proportion of rows (Figure 5.d). Of course, all of these performance improvements are problem dependent. In practice, what constitutes a "high" or "low" selectivity is difficult to determine and depends on operating and costing parameters such as network, *CPU*, and *IO* speeds and costs. This is the benefit of an algorithmic approach to distributed database design — the algorithm determines the values of "low" and "high" for each design environment and assesses the tradeoffs between replication and sophisticated operation allocation strategies.

Challenging distributed database design problems are rarely at

the extremes where it is easy to characterize retrieval and update frequencies and retrieval proportions as "high" or "low." In those cases simple rules-of-thumb, such as, "replicate when update frequency is low and retrieval proportions are high" and "use semijoins when update frequency is high and retrieval proportions are low" are sufficient for reasonable performance. The challenge occurs when there is an arbitrary mix of retrieval and update activities that must be efficiently supported. In such situations the designer must evaluate the tradeoffs between replication and the various operation allocation strategies for each query. Here automated tools such as described in this paper become valuable design aids.

To be effective over the widest range of problems, a distributed database design model must include both replication and a comprehensive set of operation allocation strategies including join order and data reduction by semijoin. In that way replication can be selected when it is efficient, and appropriate processing strategies can be used to determine a globally efficient design.

V. Summary and Future Research

We present a comprehensive distributed database design approach that treats data allocation and operating strategies in an integrated manner. Our model integrates and extends existing models of distributed database design. It includes data replication, a concurrency control mechanism, data reduction by semijoin, join node selection, and join ordering, aspects of distributed database design that are typically treated in isolation in prior work. Our solution procedure uses a nested genetic algorithm developed to solve this problem formulation. It extends both the gene structure and offspring generation components of prior algorithms. It is implemented in a workstation environment and solves realistic problems in a reasonable amount of computer time.

Using variations of an example problem, we demonstrate that replication, join node selection, reduction by semijoin and join order selection can each have a significant impact on the efficiency of a distributed database system. Replication is more

effective for retrieval intensive environments when a high proportion of rows are retrieved. Join node selection, join order, and reduction by semijoin are more effective for update intensive environments when a low proportion of rows are retrieved. When the user activities cannot be simply classified as retrieval intensive or update intensive, both replication and sophisticated operation allocation strategies contribute to the efficiency of the design. Hence we conclude that, to be most effective, a distributed database design tool is needed to support both.

Distributed database design tools such as ours provide us with insights into the effects of different data and operation allocation strategies under various conditions. Such insights can be valuable for designers of distributed databases and for organizations who must purchase or develop a distributed database management system. If, for example, an application is known to be update intensive, the designer may decide to avoid replication. This reduces the complexity of the design process, greatly simplifying the task. If a majority of applications are update intensive, the organization may decide to purchase a DBMS that supports a wide range of operation allocation strategies in its query optimizer rather than one that supports replication.

There are several areas for future research. First, the effects of data and operation allocation strategies on the efficiency of distributed database systems should be further analyzed under various conditions using real business problems. These include different types of networks with different performance parameters such as wide area networks (WAN), local area networks (LAN), and asynchronous transfer mode (ATM) networks and different types of concurrency control mechanisms such as primary copy 2PL and asynchronous updates such as *store and forward*. Second, although simulation was used to validate the model, it must be evaluated and verified in a more realistic environment. Selected solutions should be implemented and their performance measured in real organizational settings. Third, much work is needed to develop and compare alternative solution algorithms. Possible candidates include simulated annealing, partial enumeration techniques, and Lagrangian relaxation. Finally, the model itself can be extended to be more realistic. Possible extensions include the modeling of data

availability, dynamic system loads, parallel data access and different processing priorities.

Appendix 1. Operating Cost Components

Following Cornell and Yu [1989] and March and Rho [1995], the following notation is used.

Problem components:

Nodes = the number of nodes in the network.
c_{tp} = the communication cost per character from node t to node p.
s_t = unit storage cost at node t per unit time.
d_t = the cost per disk I/O at node t.
p_t = the CPU processing cost per unit at node t.
L_i = the size of file fragment i in characters.
L^M = the size of a message.
$f(k)$ = the frequency of execution of query k per unit time.
$a(k, m)$ = the file fragment used by step m of query k.
$b(k, m)$ = the second file fragment used by step m of query k for combine-fragment steps such as join and union.
$orig(k)$ = the origination node of query k.

The size of each file fragment, L_i, is calculated from the problem description parameters. The size of each temporary file is estimated from the selection and projection conditions, semijoin and join operations that produces it (see, e.g., Gardy and Peuch [1989]).

Solution components:

$copy(i, t)$ = 1 if fragment i is stored at node t, otherwise it is 0.
$op\_a(k, m, t)$ = 1 if the file fragment copy used by step m of query k is located at node t, otherwise it is 0.
$op\_b(k, m, t)$ = 1 if the second file fragment copy used by step m of query k for combine-fragment steps is located at node t, otherwise it is 0.
$node(k, m, t)$ = 1 if step m of query k is performed at node t, otherwise it is 0.
$red\_a(k, m)$ = 1 if $a(k, m)$ reduces $b(k, m)$, otherwise it is 0.
$red\_b(k, m)$ = 1 if $b(k, m)$ reduces $a(k, m)$, otherwise it is 0.

Join order is represented in the gene structure used in our genetic algorithm. It is difficult to represent this solution component using traditional decision variables.

$$COM(k, m) = \sum_{t} \sum_{p \neq t} H(k, m, t, p)\, c_{tp}$$

For message steps of retrievals,

$H(k, m, t, p) = 0$ if $t = orig(k)$
$H(k, m, t, p) = L^M * op\_a(k, m, p)$ otherwise

For join steps of base fragments when data reduction by semijoin is not used,

$$H(k, m, t, p) = (L_{a(k,m)} * op\_a(k, m, t) + L_{b(k,m)} * op\_b(k, m, t)) * node(k, m, p)$$

For joins of intermediate results and when semijoin strategies are used, the calculation of $H(k, m, t, p)$ is algorithmic, depending on the join order and the semijoin strategy. It is presented in [Rho, 1995].

For send-message steps of updates (lock request, send update, release lock operations),

$H(k, m, t, p) = 0$ if $t = orig(k)$
$H(k, m, t, p) = L^M * copy(a(k, m), p)$ otherwise

For receive-message steps of updates (lock confirmation, update confirmation operations),

$H(k, m, t, p) = 0$ if $t = orig(k)$
$H(k, m, t, p) = L^M * copy(a(k, m), p)$ otherwise.
$IO(k, m) = \sum_{t} O(k, m, t)\, d_t$

For selection and projection steps,

$$O(k, m, t) = D_{kmt} * op\_a(k, m, t)$$

where D_{kmt} is the number of disk I/Os required to process step m of query k at node t.

For join steps,

$$O(k, m, t) = (F_{a(k,m)t} * op\_a(k, m, t) + F_{b(k,m)t} * op\_b(k, m, t)) * (1 - node(k, m, t)) + (D_{kmt} + E_{a(k,m)t} * (1 - op\_a(k, m, t)) + E_{b(k,m)t} * (1 - op\_b(k, m, t))) * node(k, m, t)$$

where $F_{a(k,m)t}$ is the number of additional disk accesses needed at node t in order to send $a(k, m)$ from node t to another node after having retrieved it and $E_{a(k,m)t}$ is the number of disk access required to receive and store $a(k, m)$ at node t. Expressions to evaluate D_{kmt}, $F_{a(k,m)t}$, $F_{b(k,m)t}$, $E_{a(k,m)t}$ and $E_{b(k,m)t}$ depend on the reduction strategy selected for this query step, represented in $red\_a(k, m)$ and $red\_b(k, m)$, and on join order and reduction strategies selected for prior query steps. Detailed expressions for these are presented in [Rho, 1995].

For update requests,

$$O(k, m, t) = D_{kmt} * copy(a(k, m), t)$$

$$\mathbf{CPU(k, m) = \sum_t U(k, m, t)\, p_t}$$

For message steps,

$$U(k, m, t) = S_t * (1 - op\_a(k, m, t)) \quad \text{if } t = orig(k)$$
$$U(k, m, t) = R_t * op\_a(k, m, t) \quad \text{if } t \neq orig(k)$$

where S_t and R_t are the expected CPU units required to send and receive a message.

For selection and projection steps,

$$U(k, m, t) = W_{kmt} * node(k, m, t)$$

where W_{kmt} is the number of CPU units required to process step m of query k at node t.

For join steps,

$$U(k, m, t) = (F'_{a(k,m)t} * op\_a(k, m, t) + F'_{b(k,m)t} * op\_b(k, m, t)) * (1 - node(k, m, t)) + (W_{kmt} + E'_{a(k,m)t} * (1 - op\_a(k, m, t)) + E'$$

$_{b(k,m)t}$ * (1 - *op_b*(*k*, *m*, *t*))) * *node*(*k*, *m*, *t*)

where $F'_{a(k,m)t}$ and $E_{a(k,m)t}$ are the number of *CPU* operations required to send and receive *a*(*k*, *m*) from and to node *t*, respectively. Expressions to evaluate Wkmt, $F'_{a(k,m)t}$, $F'_{b(k,m)t}$, $E'_{a(k,m)t}$ and $E'_{b(k,m)t}$ depend on the reduction strategy selected for this query step, represented in *red_a*(*k*, *m*) and *red_b*(*k*, *m*), and on join order and reduction strategies selected for prior query steps. Detailed expressions for these are presented in [Rho, 1995].

For send-message steps of updates (lock request, send update, release lock operations),

$$U(k, m, t) = \sum_{p \neq t} copy(a(k, m), p)\ S_t \qquad \text{if } t = orig(k)$$

$$U(k, m, t) = R_t * copy(a(k, m), t) \qquad \text{if } t \neq orig(k)$$

For receive-message steps of updates (lock confirmation, update confirmation operations),

$$U(k, m, t) = \sum_{p \neq t} copy(a(k, m), p)\ R_t \qquad \text{if } t = orig(k)$$

$$U(k, m, t) = S_t * copy(a(k, m), t) \qquad \text{if } t \neq node(k)$$

For update steps,

$$U(k, m, t) = W_{kmt} * copy(a(k, m), t)$$

Appendix 2. A Nested Genetic Algorithm For Distributed Database Design

Outer Genetic Algorithm:

1. Generate initial pool of solutions:
   - 1.a. Randomly generate a feasible data allocation (to be feasible, each file (fragment) must be allocated to at least one node),
   - 1.b. Use the (inner) operation allocation genetic algorithm (see below) to allocate operations for this data allocation, thus producing a complete solution for this

data allocation,

1.c. Evaluate the cost of this solution,
1.d. Repeat until the initial solution pool is generated.

2. Iterate through successive generations:
   2.a. Probabilistically select two parent solutions from the solution pool,
   2.b. Produce a new data allocation (child) by applying crossover or mutation,
   2.c. Use the (inner) operation allocation genetic algorithm (see below) to allocate operations for this data allocation (child), thus producing a complete solution for this data allocation,
   2.d. Evaluate the cost of this solution,
   2.e. If the new solution is better than the worst solution in the solution pool, add it to the pool and remove the worst solution,
   2.f. Repeat for N generations, where N is a maximum number of iterations.

Inner Genetic Algorithm:

3. Generate initial pool of operation allocations:
   3.a. Randomly generate a feasible operation allocation for the given data allocation (to be feasible, all retrieval operations must be assigned to nodes at which the required data is stored),
   3.b. Evaluate the cost of this solution,
   3.c. Repeat until the initial operation allocation pool is generated.
4. Iterate through successive generations:
   4.a. Probabilistically select two parent solutions from the operation allocation pool,
   4.b. Produce a new operation allocation (child) by applying crossover or mutation,
   4.c. Evaluate the cost of this solution,
   4.d. If the new solution is better than the worst in the operation allocation pool, add it and remove the worst,
   4.e. Repeat for M generations, where M is a maximum number of iterations.

References

Apers, P. M. G., "Data Allocation in Distributed Database Systems," *ACM Transactions on Database Systems*, Vol. 13, No. 3, September 1988, pp. 263-304.

Apers, P. M. G., Hevner, A. R., and Yao, S. B., "Optimization Algorithms for Distributed Queries," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 1, January 1983, pp. 57-68.

Bernstein, P. A. and Chiu, D. W., "Using Semi-Joins to Solve Relational Queries," *Journal of the ACM*, Vol. 28, No. 1, January 1981, pp. 25-40.

Bernstein, P. A. and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-222.

Blankinship, R., Hevner, A. R., and Yao, S. B., "An Iterative Method for Distributed Database Optimization," *Data and Knowledge Engineering*, (21), 1997, pp. 1-30.

Brancheau, J. C., Janz, B. D., & Wetherbe, J. C., "Key Issues in Information Systems Management: 1994-95 SIM Delphi Results," MIS Quarterly, vol. 20, no. 2, pp. 225-42, 1996.

Ceri, S., Pernici, B., and Wiederhold, G., "Distributed Database Design Methodologies," *Proceedings of the IEEE*, Vol. 75, No. 5, May 1987, pp. 533-546.

Cornell, D. W. and Yu, P. S., "On Optimal Site Assignment for Relations in the Distributed Database Environment," *IEEE Transactions on Software Engineering*, Vol. 15, No. 8, August 1989, pp. 1004-1009.

Davis, L., ed., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

Dowdy, L. W. and Foster,D. V., "Comparative Models of the File Assignment Problem," *ACM Computing Surveys*, Vol. 14, No. 2, June 1982, pp. 287-314.

Eswaran, K. P., "Placement of Records in a File and File Allocation in a Computer Network," *in Information Processing '74*, Stockholm, 1974, pp. 304-307.

Epstein, R., Stonebraker, M., and Wong, E., "Query Processing in a Distributed Relational Database System," *Proceedings of ACM SIGMOD*, Austin, TX, May 1978.

Gardy, D. and Puech, C., "On the Effects of Join Operations on Relation Sizes," *ACM Transactions on Database Systems*, Vol. 14, No. 4, December 1989, pp. 574-603.

Gavish, B. and Sheng, O. R. L., "Dynamic File Migration in Distributed

Computer Systems," *Communications of the ACM*, Vol. 33, No. 2., February 1990, pp. 177-189.

Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

Hevner, A. R., *The Optimization of Query Processing on Distributed Database Systems*, Ph.D. Thesis, Purdue University, 1979.

Hevner, A. R. and Yao, S. B., "Query Processing in Distributed Database Systems," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, May 1979, pp. 177-187.

Johansson, J. M., "Impact of High-Speed Wide Area Network Response Time Dynamics on Distributed Database Design," PhD dissertation in Information and Decision Sciences Dept. Minneapolis: University of Minnesota, 1999.

Kleinrock, L., *Queuing Systems: Theory*, John Wiley & Sons, 1975.

Lee, H. and Sheng, O. R. L., "A Multiple Criteria Model for the Allocation of Data Files in a Distributed Information Systems," *Computers and Operations Research*, Vol. 21, 1992, pp. 21-33.

Lohman, G. M., Mohan, C., Haas, L. M., Daniels, D., Lindsay, B. G., Selinger, P. G., and Wilms, P.F., "Query Processing in R*," in Kim, W. et al. (eds.) *Query Processing in Database Systems*, Spring-Verlag, Berlin, 1985, pp. 31-47.

March, S. T. and Rho, S., "Allocating Data and Operations to Nodes in Distributed Database Design," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 7, No. 2, April 1995, pp. 305-317.

Mishra, P. and Eich, M. H., "Join Processing in Relational Databases," *ACM Computing Surveys*, Vol. 24, No. 1, March 1992, pp. 63-113.

Ozsu, M. and Valduriez, P., "Distributed Database Systems: Where Are We Now?" *IEEE Computer*, August 1991a, pp. 68-78.

Ozsu, M. and Valduriez, P., *Principles of Distributed Database Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991b.

Ram, S. and Marsten, R. E., "A Model for Database Allocation Incorporating a Concurrency Control Mechanism," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 3, 1991, pp. 389-395.

Ram, S. and Narasimhan, S., "Allocation of Databases in a Distributed Database System," *Proceedings of the 11th International Conference on Information Systems*, December 1990, pp. 215-230.

Ram, S. and Narasimhan, S., "Database Allocation in a Distributed Environment: Incorporating a Concurrency Control Mechanism and Queuing Costs," *Management Science*, Vol. 40, No. 8, August 1994, pp. 969-983.

Rho, S., *Distributed Database Design: Allocation of Data and Operations to Nodes in Distributed Database Systems*, Unpublished Ph.D.

Thesis, University of Minnesota, May 1995.

Rho, S. and March, S. T. "Designing Distributed Database Systems for Efficient Operation," *Proceedings of the 16th International Conference on Information Systems*, December 1995, pp. 237-253.

Rho, S. and March, S. T., "Optimizing Distributed Join Queries: A Genetic Algorithm Approach," *Annals of Operations Research*, Vol. 71, 1997, pp. 199-228.

Ricciuti, M., "DBMS Vendors Chase Sybase for Client/Server," *Datamation*, Vol. 39, July 1, 1993, pp. 27-28.

Richter, J., "Distributing Data," Byte, June 1994, pp. 139-148.

Syswerda, G., "Uniform Crossover in Genetic Algorithm," *Proceedings of the 3rd International Conference on Genetic Algorithms*, 1989, pp. 2-9.

The, L., "Distribute Data Without Choking the Net," *Datamation*, Vol. 40, January 7, 1994, pp. 35-36.

Teorey, T. J., *Database Modeling and Design*, Morgan Kaufmann, San Mateo, CA, 1990.

Yoo, H. and Lafortune, S., "An Intelligent Search Method for Query Optimization by Semijoins," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 1, No. 2, June 1989, pp. 226-237.

Yu, C. T. and Chang, C. C., "Distributed Query Processing," *ACM Computing Surveys*, Vol. 16, No. 4, December 1984, pp. 399-433.