# A Nested Genetic Algorithm for Distributed Database Design

Sangkyu Rho
Salvatore T. March

Information and Decision Science Department
University of Minnesota

## Abstract

*Distributed database design is a difficult and complex process involving two major, interrelated problems. First, data must be allocated to nodes in the network. Second, given such an allocation, data must be efficiently accessed, processed, and possibly communicated to meet the retrieval and update requirements of the users. Both of these problems can be formulated as constrained, integer, optimization problems; both of which are NP-hard. Genetic algorithms provide an efficient search method for problems of this type. We present a nested genetic algorithm that iteratively allocates data to nodes and determines where to perform access and processing operations to efficiently meet a specified set of retrieval and update requirements.*

## 1. Introduction

With the emergence of relatively inexpensive, high-capacity communications capabilities, geographically distributed systems have become an integral part of many computer applications. Distributed systems can yield significant management and organizational advantages over centralized systems [16]. Judicious placement of data and processing capabilities can significantly reduce operating costs and response time. Inappropriate placement of data or poor choices of data access, however, can result in high cost and poor system performance [20].

The design of efficient distributed databases is a complex and difficult process involving two interrelated problems, data allocation and operation allocation [1]. *Data allocation* defines what data is allocated to each node in the network (see [9] for a survey of methods). To enhance retrieval efficiency, the same data can be redundantly allocated to multiple nodes. Such redundancy increases update costs. *Operation allocation* defines where retrieval and processing operations will be performed. Retrieval operations must, of course, be allocated to a node containing the required data. Processing operations can be allocated to any node. However, if the data is not located at the processing node, it must be sent over the communication network. Update operations must be done at all nodes containing a copy of the effected data.

Data and operation allocation are interdependent problems and must be solved simultaneously. The optimal set of file copies and their optimal allocation depend on the processing schedules of all queries accessing these files (i.e., the operation allocation); the optimal processing schedules depend on where file copies are located (i.e., the data allocation). Hence, to be effective a distributed database design approach must comprehensively treat both data and operation allocation as a unified whole.

Numerous mathematical models have been developed for distributed database design (see, e.g., [1, 2, 19]). One of the difficulties facing researchers in this area is tractability. Even simplistic models treating only one of the above problems have been shown to be NP-hard [10, 13]. Therefore, we need extremely efficient and adaptable procedures to address distributed database design problems of realistic size. One promising approach is the genetic algorithm (GA) (see, e.g., [11, 15]). We have developed a genetic algorithm for a comprehensive model of the distributed database design problem [19]. In this paper, we describe that algorithm and analyze its effectiveness.

## 2. Genetic algorithms

Genetic algorithms (GA) are a class of robust and

**Figure 1. Example Solution Pool with Performance and Fitness**

| Solution | Performance (communication volume) | Fitness | Selection Probability |
|---|---|---|---|
| 1111 1101 1010 0100 0100 | 9M characters / day | .85 | .170 |
| 0001 0100 0010 1000 1100 | 14M characters / day | .77 | .154 |
| 1011 1101 0110 1101 0101 | 7M characters / day | .88 | .176 |
| 0100 0010 1001 0111 0100 | 12M characters / day | .80 | .160 |
| 0010 0101 1000 0100 0101 | 13M characters / day | .78 | .156 |
| 1010 0101 1111 1111 0111 | 5M characters / day | .92 | .184 |
| Total Pool | 60M characters / day | 5.00 | 1.000 |

efficient search methods based on the concept of the adaptation in natural organisms. They have been successfully applied to complex problems in diverse fields, including the traveling salesperson problem [17], facility layout design [22], rule induction [3, 12], communication network design [5, 6], and VLSI cell placement [21].

The basic ideas of GA are: (1) a representation of solutions, typically in the form of bit strings, likened to genes in a living organism, (2) a pool of solutions likened to a population or generation of living organisms, each having a genetic make-up; (3) a Darwinian notion of "fitness," which governs the selection of parents who will produce offspring in the next generation; (4) genetic operators, which derive the genetic make-up of an offspring from that of its parents (and possible random "mutation"); and (5) survival of the fittest where the least fit solutions are removed from the solution pool at each generation (do not survive into the next generation) [7].

To illustrate these components, consider a file allocation problem with five files and four nodes (ignore, for the moment, the operation allocation problem). A solution can be represented by five sets of four bits each, one set for each file (alternately a solution could be represented by four sets of five bits each, one set for each node). The four bits in each set represent where the corresponding file is stored. For example, the set 1000 stores the file at node 1, the set 1100 stores the file at nodes 1 and 2, etc. The solution: (0011 0101 1110 0111 0100) stores file 1 at nodes 3 and 4; file 2 at nodes 2 and 4; file 3 at nodes 1, 2, and 3; file 4 at nodes 2, 3, and 4; and file 5 at node 2. Thus, for this problem the gene structure is defined as five sets of four bits each.

A genetic algorithm begins by randomly generating an initial pool of solutions (i.e., the *population*). The poolsize is a parameter of a genetic algorithm. The pool should be large enough to insure a reasonable sample of the actual solution space, but not so large as to make the algorithm approach exhaustive enumeration. A poolsize on the order of 100 is typically sufficient for a solution space in the billions. The five file, four node file allocation problem has 1,048,576 (i.e., $2^{20}$) possible solutions (not all of which will be feasible). Figure 1 shows a randomly generated pool of size 6 for this problem (this poolsize is probably not sufficient to adequately simulate natural genetics).

During each iteration, called a *generation*, the solutions in the pool are evaluated using some measure of fitness or performance. In the file allocation problem, solutions are typically evaluated in terms of operating cost or response time for a given set of retrieval and update requirements and a given system configuration (e.g., network design and unit costs for communication, storage, and I/O).

Consider minimum communication volume as the evaluation criterion. Suppose that the network is fully connected and that each file is 1 million characters and is accessed once per day from each node. For illustrative purposes, ignore communication volume due to update (we note that the optimal solution is to replicate all files at all nodes). Communication is required if a file is not stored at a node. Hence, for the first solution, (1111 1101 1010 0100 0100), communication is required to send files 4, and 5 to node 1 (2 * 1M characters); file 3 to node 2 (1M characters); files 2, 4, and 5 to node 3 (3 * 1M characters); and files 3, 4, and 5 to node 4 (3 * 1M characters). Thus the performance of this solution is 9M characters per day. The performance can be similarly calculated for each of the solutions in the pool (see Figure 1).

Since we are attempting to minimize the volume of data communicated, the fitness of a solution can be calculated as one minus the solution's contribution to volume transferred. The first solution, for example, contributes 9/60 of the volume of data transferred, its fitness is (1 - 9/60), or .85. Fitness for each solution is similarly calculated (again, see Figure 1).

After evaluating the fitness of each solution in the pool, some of the solutions are selected to be parents. The probability of any solution being selected is proportional to its fitness. For example, as illustrated in Figure 1, the probability of selecting the first solution is .170 (.85/5.00). Parents are paired and genetic operators applied to produce new solutions, called *offspring*. A new generation is formed by selecting solutions (parents and offspring) based on their performance, so as to keep the poolsize constant. Solutions could be selected based strictly on performance, i.e., keeping only the best solutions. Alternately, solutions could be selected probabilistically.

The genetic operators commonly used to produce offspring are crossover and mutation. Crossover is the primary genetic operator. It operates on two solutions (parents) at a time and generates offspring by combining segments from both parents. A simple way to achieve crossover is to select a cut point at random and produce offspring by concatenating the segment of one parent to the left of the cut point with that of the other parent to the right of the cut point. A second offspring can be produced by combining the opposite segments. Alternately, genes for an offspring can be randomly selected from each parent.

Mutation generates a new solution by independently modifying one or more gene values of an existing solution, selected at random. It serves to guarantee that the probability of searching a particular subspace of the solution space is never zero. Mutation rates are typically quite low. Goldberg [11] suggests mutation rates on the order of .001. When the mutation rate is very high, generic algorithms approach random search.

**Figure 2.** Crossover Producing Two Offspring From Two Parents

| | |
|---|---|
| Parent 1: | 1111 1101 1010 0100 0100 |
| Parent 2: | 1010 0101 1111 1111 0111 |
| | \| crossover point |
| Offspring 1: | 1111 1101 1111 1111 0111 |
| Offspring 2: | 1010 0101 1010 0100 0100 |

Continuing the five file, four node data allocation problem, suppose the first and last solutions are selected to be parents. Figure 2 illustrates the production of two offspring using crossover with a single crossover point after the eighth bit (the crossover point is typically randomly selected for each "mating"). The two offspring require 2M and 12M characters of data transfer per day, respectively. If no other offspring were generated, and best performance was used to form a new generation, then the next generation would include both of these offspring in the pool and solutions 2 and 5 (with 14M and 13M characters of data transfer, respectively) would be eliminated.

The best solution in the new generation transfers 2M characters (as opposed to 5M in the prior generation). The total number of characters of data transferred by all solutions in the pool is reduced from 60M to 47M. Hence, the performance of the best solution and the "overall fitness" of the pool of solutions have improved.

A solution is mutated simply by choosing a bit at random and switching it from its current value to its complement (or another random value if genes are not binary). For example, suppose the first solution in Figure 1 was selected for mutation (i.e., 1111 1101 1010 0100 0100) and the second bit was selected at random. The mutation would be 1011 1101 1010 0100 0100, and the performance of the mutation would be worse than the original solution (i.e., 10M characters versus 9M characters). If, on the other hand, bit 7 was randomly selected, then the mutation would be 1111 1111 1010 0100 0100, and its performance would be better than the original (i.e., 8M characters versus 9M characters).

Although greatly simplified, the above example provides insight into why genetic algorithms are effective. As crossover combines solutions, a number of partial solutions, termed *schemas*, having good performance begin to emerge in multiple solutions. In Figure 2, Parent 1 has a good schema for files 1 and 2 (redundantly stored at nearly every node), while Parent 2 has a good schema for files 3, 4 and 5. The offspring generated when the first part of Parent 1 is combined with the second part of Parent 2 has better performance then either parent. Thus the good parts (i.e., schema) of each parent will be preserved.

Generally, good schemas are not so simply defined. They often involve subsequences of genes. For example, Parents 1 and 2 in Figure 2 share the schema 1*1* *101 1*1* *1** 01**, where * indicates that a bit's value is unspecified (of course, they share many other schemas, such as 1*1* **** **** **** **** and **** *101 **** **** ****). All of the solutions in Figure 1 share the schema **** **** **** **** *1**.

Parents with above average performance are expected to contain some number of good schemas. Due to the stochastic selection process, such parents are likely to produce more offspring than those with below average performance (which are expected not to contain as many good schemas). Over successive iterations (generations), the number of good schemas represented in the pool tends to increase, and the number of bad schemas tends to decrease. Therefore, the average performance of the pool tends to improve.

The power of a genetic algorithm stems from what is called *implicit parallelism* [11, 15]. The implicit parallelism theorem sets a lower bound of an $N^3$ speedup over exhaustive enumeration where N is population size [15]. DeJong and Spears [8] empirically demonstrated that the theoretical lower bound holds in NP-complete problems. In a genetic algorithm, implicit parallelism is achieved in the following way. Each solution is a sample for all the solutions sharing any of its many schemas. For example, the solution 1101 represents 1***, *1**, 11**, and so on. Evaluating a solution provides information about *all* solutions sharing *any* of its schemas. Thus, a genetic algorithm that evaluates a relatively small pool of solutions actually samples a vastly larger portion of the solution space.

The final issue with genetic algorithms is stopping rules. Theoretically iteration through successive generations could continue indefinitely. The most common stopping rule for genetic algorithms is a maximum number of iterations. Other possible stopping rules include: (1) limits on the number of generations with no improvement in the best solution or no improvement in the total fitness of the pool and (2) limits on the difference between the fitness of the worst and best solutions in the pool. These stopping rules attempt to recognize when the genetic material in the pool is so similar that new solutions will only be produced by mutation.

## 3. Distributed database design

In general, distributed database design involves three steps [19]. First, the data is partitioned into a set of file fragments for allocation. File fragments are typically defined based on the selection and projection criteria of the set of known queries [1]. Second, each query is decomposed into a set of query steps or operations, each of which references at most two file fragments [4]. Query steps include communication steps (i.e., sending messages and result files) as well as data retrieval and manipulation steps (i.e., select, project, join, union, and final processing). Third, file fragments and query steps are allocated to nodes. Of course, communication steps result in zero cost if retrieval and manipulation steps are performed at the node containing the data needed for that step.

In this section we briefly describe the model proposed by March and Rho [19] for distributed database design. We then describe a nested genetic algorithm for its solution.

In the model proposed by March and Rho [19], the following sets of zero-one decision variables are used:

$X_{it}$ = 1 if file fragment i is stored at node t
0 otherwise

$Z_{kit}$ = 1 if query k uses file fragment i at node t
0 otherwise

$Y_{kmt}$ = 1 if step m of query k is done at node t
0 otherwise

The $X_{it}$ variables represent the data allocation (with redundancy), the $Z_{kit}$ and $Y_{kmt}$ variables represent the operation allocation.

The objective is to minimize system operating costs, including communication, disk I/O, CPU processing, and storage:

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

where f(k,j) is the frequency of execution of query k originating at node j per unit time, COM(k,j,m), IO(k,j,m), and CPU(k,j,m) are the respective costs of communication, disk I/O, and CPU processing time for step m of query k originating at node j, and STO(t) is the cost of storage at node t per unit time.

Resource constraints as well as intrinsic problem constraints are considered. The intrinsic problem constraints are:

$\sum_t X_{it} \geq 1$ for all file fragments, i = 1, 2, ... , number of fragments (all file fragments must be stored at one or more nodes)

$Z_{kit} \leq X_{it}$ for all queries, k = 1, 2, ... , number of queries for all file fragments, i = 1, 2, ... , number of fragments for all nodes, t = 1, 2, ... , number of nodes (a file fragment cannot be accessed from a node unless it is stored at that node)

$\sum_t Y_{kmt} = 1$ for all queries, k = 1, 2, ... , number of queries for all steps m, m = 1, 2, ... , number of steps for query k (all query steps must be processed at some node).

Resource constraints restrict disk I/O, CPU processing, and storage capacity at each node and the communication capacity of each link. Expressions for each of the cost components and constraints are described in [19] and summarized in Appendices 1 and 2.

## 4. A nested genetic algorithm

As discussed in Section 1, data allocation and operation allocation are interrelated problems, each of which is NP-complete. Apers [1] developed an integrated solution method which sequentially optimizes query plans and then allocates fragments. Blankinship et al. [2] developed an iterative method which alternates between distributed query optimization and distributed data allocation optimization. However, both of these approaches can result in locally optimal solutions and can become intractable as problem size increases. Furthermore, neither approach considers data replication which further increases the size of the solution space.

To address the tractability problem, we developed a genetic algorithm-based solution method. A genetic algorithm was chosen for several reasons. First, genetic algorithms have been successfully applied to similar complex, combinatoric, real-world problems (see, for example, [8, 11, 14]). Second, genetic algorithms are robust in that they work well even in discontinuous, multimodal, noisy search spaces [11]. Third, genetic algorithms result not only in a "best" solution, but also in a pool of good solutions.

This last point is important since the set of solutions in the final pool provides significant intuition into the effects of design alternatives. That is, solutions represent "good" schemas (partial solutions) that the designer should be able to recognize from the final pool. For example, if all solutions in the final pool store a given file fragment at a particular node, the designer would be reasonably confident that it is important to store that file at that node.

Our genetic distributed database design algorithm contains a genetic algorithm within a genetic algorithm. The outer genetic algorithm addresses file allocation. The inner genetic algorithm addresses operation allocation.

File allocation solutions, i.e., the $X_{it}$ variables, are represented as described in Section 2 and illustrated in Figures 1 and 2. Each gene has (I * T) bits where

I is the number of file fragments and T is the number of nodes (i.e., I sets of T bits each). Operation allocation solutions, i.e., the $Z_{kit}$ and $Y_{kmt}$ variables, are represented simply by a vector with a position for each query step (operation). Each value in the vector is the node at which the query step is performed. Retrieval operations can only be done at nodes at which the required data is stored. If an offspring assigns a retrieval operation to a node at which the data is not stored, that offspring has poor performance, and will likely be eliminated from the pool. Other operations (such as joins and unions) can be performed at any node. Data is sent to the node at which these operations are performed as necessary.

Our nested genetic algorithm operates as follows:

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 (nested) operation allocation genetic algorithm (see below) to allocate operations for this data allocation, thus producing a complete solution for this data allocation,
   - 1.c. Repeat step 1 until the entire solution pool is generated (solution poolsize is an input parameter).

2. Iterate through successive generations:
   - 2.a. Probabilistically select two parent solutions from the solution pool (the selection probability is equal to the fitness of the solution divided by the total fitness of the pool, where fitness is inversely related to cost),
   - 2.b. Produce a new data allocation by applying crossover or mutation (a random assignment of parent genes to offspring is used for crossover; the mutation rate is an input parameter),
   - 2.c. Use the (nested) operation allocation genetic algorithm (see below) to allocate operations for this data allocation (offspring), thus producing a complete solution for this data allocation,
   - 2.d. 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.e. Repeat until the worst solution is within x% of the best (where x is an input parameter).

The genetic algorithm to allocate operations for a given data allocation, used in steps 1.b. and 2.c., is similar:

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 step 3 until the entire operation allocation pool is generated (operation allocation poolsize is an input parameter),

4. Iterate through successive generations:
   - 4.a. Probabilistically select two parent solutions from the operation allocation pool (the selection probability is equal to the fitness of the solution divided by the total fitness of the pool, where fitness is inversely related to cost),
   - 4.b. Produce a new operation allocation by applying crossover or mutation (a random assignment of parent genes to offspring is used for crossover; the mutation rate is an input parameter),
   - 4.c. Evaluate the cost of this solution (offspring),
   - 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 until the worst solution in the operation allocation pool is within y% of the best (where y is an input parameter).

The genetic algorithm is written in C++ and runs in a MS-DOS or UNIX environment. To test the genetic algorithm we solved a series of 13 small problems (3 nodes, 3 file fragments, and 12 to 18 query steps) with the pool size of 40 and the mutation rate of 0.005. We then compared the result to the optimal solution obtained by exhaustive enumeration. The genetic algorithm found the optimal solution for all 13 problems. The run time for the genetic algorithm was four to eight minutes on an IBM-compatible PC with a 33Mhz 80386 processor.

We also solved a larger problem (4 nodes, 9 file fragments, and over 100 query steps) and obtained a reasonable solution in approximately 11 hours of computing time. This was reduced to under one hour on a UNIX Sun workstation. We are currently doing a series of experiments with this problem varying the poolsize, mutation rate, and stopping rules for both genetic algorithms. Run times on the UNIX Sun Workstation range from under one hour to over fifteen hours. As expected, larger poolsizes and high iteration-based stopping rules lead to significant increases in run time. The best known solution was found in four hours with a poolsize of 300 and a mutation rate of 0.01. Our goal is to understand the relationship between problem parameters, poolsize, mutation rate, and stopping rules.

## 5. Conclusion and Future Research

We have developed a nested genetic algorithm to solve a comprehensive formulation of the distributed database design problem. The outer genetic algorithm addresses data allocation while the inner genetic algorithm addresses operation allocation. Advantages of a genetic algorithm include its robustness and its efficiency. The evaluation criteria for a solution (i.e., the objective function to be optimized) can be extremely complex including nonlinearities and discontinuities, characteristics that render traditional optimization techniques ineffective.

Future research will progress in several directions. First, we are analyzing the effects of poolsize and mutation rate on the run time and performance of the algorithm. We hypothesize that the performance curve of the algorithm is S-shaped with respect to poolsize. That is, until the poolsize reaches a sufficient size to contain a "reasonable" amount of genetic material, the effects of increasing poolsize are small. Once the poolsize reaches this threshold, performance increases dramatically until the pool becomes "saturated" with genetic material. At this point increases in the poolsize again have minimal effect on performance. We are attempting to analyze various problem parameters to determine good poolsizes.

Initial findings also indicate that as mutation is most significant when the poolsize is "small." At larger poolsizes, mutation is insignificant. This is reasonable as the effect of mutation is to expand the genetic material in the pool. If the pool is sufficiently large, it probably already contains all the necessary genetic material to find very good, if not optimal, solutions.

A second direction for future research is to investigate the effects of different crossover methods and stopping rules. Goldberg [11] indicates that using a single point crossover and generating two offspring per pair of parents insures that "good" schemas are retained. We have implemented "random" crossover and generate only one offspring. Our stopping rule compares the best and worst solutions in the pool. We will analyze different stopping rules such as maximum iterations to determine the effect on run time and performance. Similarly, since genetic algorithms are probabilistic in nature we will analyze the relationship between poolsize and number of iterations. In addition, since mutation tends to add new genetic material we will investigate the effects of increasing the mutation rate when the pool becomes too "similar."

Finally, future research will compare the performance of our genetic algorithm with alternate algorithms. As mentioned above, the genetic algorithm can treat more realistic cost functions than standard optimization approaches. Furthermore, it can easily enforce constraints, a difficulty with strictly numerical approaches. However there are several other approaches that could compete with genetic algorithms. Among them are: branch and bound algorithms, simulated annealing, switching heuristics, and randomized hill-climbing approaches.

## References

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

2. Blankinship, R., Hevner, A. R., and Yao, S. B., "An Iterative Method for Distributed Database Design," Proceedings of the Seventeenth International Conference on Very Large Data Bases, Barcelona, Spain, September 1991, pp. 389-400.

3. Chung, H. M. and Silver, M. S., "Rule-Based Expert Systems and Linear Models: An Empirical Comparison of Learning-By-Example Methods," Decision Sciences, Vol. 23, No. 3, May/June 1992, pp. 687-707.

4. 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.

5. Coombs, S. and Davis, L., "Genetic Algorithms and Communication Link Design: Constraints and Operators," Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, July 28-31, 1987.

6. Davis, L. and Coombs, S., "Genetic Algorithms and Communication Link Design: Theoretical Considerations," Proceedings of the 2nd International Conference on Genetic Algorithms, Cambridge, MA, July 28-31, 1987.

7. De Jong, K. A., "Genetic-Algorithm-Based Learning," in Kodratoff, Y. and Michalski, R. S. (eds.), Machine Learning, Morgan Kaufmann Publishers, 1990, pp. 611-638.

8. De Jong, K. A. and Spears, W. M., "Using Genetic Algorithms to Solve NP-Complete Problems," Proceedings of the 3rd International Conference on Genetic Algorithms, June 4-7, 1989, Morgan Kaufmann Publishers.

9. 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.

10. 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.

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

12. Greene, D. P., "Automated Knowledge Acquisition: Overcoming the Expert System Bottleneck," Proceedings of the 8th International Conference on Information Systems, Pittsburgh, PA, December 6-9, 1987.

13. Hevner, A., The Optimization of Query Processing on Distributed Database Systems, PhD Thesis, Purdue University, 1979.

14. Holland, J. H., "Genetic Algorithms," Scientific American, July 1992, pp. 66-72.

15. Holland, J. H., Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, 1975.

16. King, J. L., "Centralized versus Decentralized Computing: Organizational Considerations and Management Options," ACM Computing Surveys, Vol. 15, No. 4, December 1983, pp. 319-349.

17. Lu, Y., Solving Combinatorial Optimization Problems by Simulated Annealing, Genetic Algorithms, and Neural Networks, Unpublished MS thesis, University of Minnesota, 1991.

19. March, S. T. and Rho, S., "Allocating Data and Operations to Nodes in Distributed Database Design," IEEE Transactions on Knowledge and Data Engineering, to appear.

20. Ozsu, M. and Valduriez, P., Principles of Distributed Database Systems, Prentice-Hall, Inc., 1991.

21. Shahookar, K. and Mazumder, P., "VLSI Cell Placement Techniques," ACM Computing Surveys, Vol. 23, No. 2, June 1991, pp. 143-220.

22. Tam, K. Y., "Genetic Algorithms, Function Optimization, and Facility Layout Design," European Journal of Operational Research, Vol. 63, No. 2, December 10, 1992, pp. 322-346.

## Appendix 1. Components of the Objective Function

**Communication Costs:** COM(k,j,m) =

$\sum_t Z_{k,a(k,m),t} L^M c_{jt}$ for message steps of retrieval

$\sum_t \sum_p Z_{k,a(k,m),t} Y_{kmp} L_{a(k,m)} c_{tp}$ for selection and projection steps

$\sum_t \sum_p Y_{kmp} c_{tp} (Y_{k(m-3)t} L_{a(k,m)} + Y_{k(m-1)t} L_{b(k,m)})$ for combine fragment steps

$\sum_t Y_{k(m-1)t} L_{a(k,m)} c_{jt}$ for final steps

$\sum_t X_{a(k,m)t} L^M c_{jt}$ for message steps of update

where a(k,m) and b(k,m) are the file fragments referenced by step m of query k; $L_i$ and $L^M$ are the size of file fragment i and the size of a message, respectively; and $c_{tp}$ is the communication cost per character from node t to p.

**Disk I/O Costs:** IO(k,j,m) = $\sum_t O(k,j,m,t)\, d_t$

where O(k,j,m,t) is the disk I/O load at node t due to step m of query k origination at node j and $d_t$ is the cost per disk I/O at node t. O(k,j,m,t) for each step is defined as follows:

O(k,j,m,t) = $Y_{kmt} D_{kmt} + (1-Y_{kmt}) Z_{k,a(k,m),t} F_{a(k,m)t} + Y_{kmt} (1-Z_{k,a(k,m),t}) E_{a(k,m)t}$ for selection and projection steps

$Y_{kmt} D_{kmt} + (1-Y_{kmt}) (Y_{k(m-3)t} F_{a(k,m)t} + Y_{k(m-1)t} F_{b(k,m)t}) + Y_{kmt} ((1-Y_{k(m-3)t}) E_{a(k,m)t} + (1-Y_{k(m-1)t}) E_{b(k,m)t})$ for combine fragment steps

$Y_{k(m-1)t} F_{a(k,m)t}$ if $j \neq t$, and
$(1-Y_{k(m-1)t}) E_{a(k,m)t}$ if $j = t$ for final steps

$X_{a(k,m)t} D_{kmt}$ for update steps

where $D_{kmt}$ is the number of disk I/Os required to process step m of query k at node t, $F_{a(k,m)t}$ is the number of disk I/Os needed at node t to send a(k,m) from node t to another node, and $E_{a(k,m)t}$ is the number of disk I/Os required to receive and store a(k,m) at node t.

**CPU Costs:** CPU(k,j,m) = $\sum_t U(k,j,m,t)\, p_t$

where U(k,j,m,t) is the number of CPU processing units expended at node t for local processing and communication for step m of query k originating at node j and $p_t$ is the CPU processing cost per unit. U(k,j,m,t) for each step is defined as follows:

U(k,j,m,t) = $(1-Z_{ka(k,m)t}) S_t$ if $j = t$, and
$Z_{ka(k,m)t} R_t$ if $j \neq t$ for message steps of retrieval

$Y_{kmt} W_{kmt} + (1-Y_{kmt}) Z_{k,a(k,m),t} F'_{a(k,m)t} + Y_{kmt} (1-Z_{k,a(k,m),t}) E'_{a(k,m)t}$ for selection and projection steps

$Y_{kmt} W_{kmt} + (1-Y_{kmt}) (Y_{k(m-3)t} F'_{a(k,m)t} + Y_{k(m-1)t} F'_{b(k,m)t}) + Y_{kmt} ((1-Y_{k(m-3)t}) E'_{a(k,m)t} + (1-Y_{k(m-1)t}) E'_{b(k,m)t})$ for combine fragment steps

$(1-Y_{k(m-1)t}) E'_{a(k,m)t}$ if $j = t$, and
$Y_{k(m-1)t} F'_{a(k,m)t}$ if $j \neq t$ for final steps

$\sum_{p \neq t} X_{a(k,m)p} S_t$ if $j = t$, and

| | | |
|---|---|---|
| $X_{a(k,m)t} R_t$ | if $j \neq t$ | for send-message steps of update |
| $\sum_{p \neq t} X_{a(k,m)p} R_t$ | if $j = t$, and | |
| $X_{a(k,m)t} S_t$ | if $j \neq t$ | for receive-message steps of update |
| $X_{a(k,m)t} W_{kmt}$ | | for update steps |

where $W_{kmt}$ is the number of CPU units required to process step m of query k at node t; $S_t$ and $R_t$ are the expected CPU units required to send and receive a message; and $F'_{a(k,m)t}$ 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.

**Data Storage Costs:** $STO(t) = G(t)\, s_t$

where $G(t) = \sum_i X_{it} L_i$ and $s_t$ is the unit storage cost per unit time at node t.

## Appendix 2. Resource Constraints

### Disk I/O Capacity Constraints

$\sum_k \sum_j f(k,j) \sum_m O(k,j,m,t) \leq UIO(t)$ for each node, $t = 1, 2, \ldots$, number of nodes

where UIO(t) is the disk I/O capacity at node t.

### CPU Capacity Constraints

$\sum_k \sum_j f(k,j) \sum_m U(k,j,m,t) \leq UCPU(t)$ for each node, $t = 1, 2, \ldots$, number of nodes

where UCPU(t) is the CPU processing capacity at node t.

### Storage Capacity Constraints

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

where US(t) is the storage capacity at node t.

### Communication Link Capacity Constraints

$\sum_k \sum_j f(k,j) \sum_m H(k,j,m,t,p) \leq UL(t,p)$ for each link (t,p), $t = 1, 2, \ldots$, no of nodes; $p = 1, 2, \ldots$, no of nodes.

where H(k,j,m,t.p) is the amount of communication on the link connecting nodes t and p due to step m of query k originating at node j and UL(t,p) is the communication capacity of link from node t to p. H(k,j,m,t,p) for each step is defined as follows:

| | | | |
|---|---|---|---|
| $H(k,j,m,t,p) =$ | $Z_{ka(k,m)p} L^M$ | if $j = t$ | |
| | $Z_{ka(k,m)t} L^M$ | if $j = p$ | |
| | $0$ | otherwise | for message steps of retrieval |
| | $L_{a(k,m)} (Z_{ka(k,m)t} Y_{kmp} + Z_{ka(k,m)p} Y_{kmt})$ | | for selection and projection steps |
| | $L_{a(k,m)} (Y_{k(m-3)t} Y_{kmp} + Y_{k(m-3)p} Y_{kmt}) + L_{b(k,m)} (Y_{k(m-1)t} Y_{kmp} + Y_{k(m-1)p} Y_{kmt})$ | | for combine fragment step |
| | $Y_{k(m-1)p} L_{a(k,m)}$ | if $j = t$ | |
| | $Y_{k(m-1)t} L_{a(k,m)}$ | if $j = p$ | |
| | $0$ | otherwise | for final steps |
| | $X_{a(k,m)p} L^M$ | if $j = t$ | |
| | $X_{a(k,m)t} L^M$ | if $j = p$ | for both send-message steps and |
| | $0$ | otherwise | receive-message steps |