DATA ALLOCATION

IN A

DISTRIBUTED DATABASE ENVIRONMENT

by

Kimberley Ann Johnson

B. S., Western Illinois University, 1979

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

Major Professor

1.4 Report Organization

This report is organized into 4 chapters. Chapter 2 provides an overview of the problems encountered in data distribution and reviews the relevant literature in this area. Chapter 3 provides an overview of the project's design. Chapter 4 contains an extensive example to demonstrate the functionality and use of the tool. Chapter 5 gives concluding remarks and suggests possible extensions.

CHAPTER 2

THE DATA DISTRIBUTION PROBLEM

2.1 Objective

The objective of this chapter is to summarize the many problems addressed in the literature concerning data distribution. A high level understanding of these problems and their interdependencies will familiarize the reader with the complexity these problems introduce into the distributed database design, even when addressed individually. A great deal of research has been done to address different parts of the data distribution problem. Some of the significant work in these areas will be reviewed. To date however, there has been no overall solution that successfully addresses all of the problems under one combined methodology. Further research continues to look for optimal solutions to these issues.

2.2 Why Distribute ?

In a distributed environment there are clear benefits to be derived in distributing data. One obvious advantage is the ability to store the data at the location(s) where it is most frequently used. This will achieve a faster response time and reduce communication costs in a query intensive application. A second advantage in distributing data is the potential to store what might be a very large database on smaller machines through partitioning or splitting the data (partitioning will be reviewed in detail in a later section). As a final example, the distribution of data increases the overall reliability of the system by the simple fact that all data is not stored on one machine (i.e., subject to single site failure) [ROTH81]. Each of these points demonstrate clear advantages, but maximizing these benefits requires important tradeoffs.

2.3 Distribution Schemes and Associated Problems

Consider the optimal allocation of a file in a distributed environment. Intuition would tell you to

store the data where it is most frequently used. If each site owned and used the data exclusively (i.e., no other site needed any of this data) there would be no design problem. All data updates and queries at this node could be handled locally. In reality though, this is not the case as applications have data that is shared among several users. These applications may also have strict requirements as to response time, reliability and consistency of the data they require. For this reason, careful analysis of the distribution scheme to be used is required.

Distribution schemes generally fall into two main categories: partitioned systems and replicated systems [DRAF80].

2.3.1 Partitioned Systems

Partitioning is the process of "assigning a local object (relation) from the logical schema of the database to several physical objects (files) stored in the database" [BRAY81]. A pure partitioned system has no duplication or replication of data items. Vertical partitioning (or partitioning by structure) divides the data by columns or attributes. Application of vertical partitioning would be desirable in cases where only certain attributes of the logical record are needed at locations. An example taken from [DRAF80] will help illustrate this point. This example considers a relation for orders of parts as follows:

ORDERS (CUST #,CNAME,PART NO,PART DESC,QUANTITY ORDERED)

One site may only be concerned with the PART_NO and PART_DESC, while another site may maintain the CUST_NO and CUST_NAME. In this instance there are benefits derived from partitioning the data vertically, as each user can locally control the information with which they are directly concerned.

Horizontal partitioning divides a relation by occurrence or tuple. This type of partitioning is valuable in cases where files can be distributed based upon given data values. Again, using the

ORDERS relation, each site may need all information in the relation but may only deal with one type of part. In this case all tuples associated with a given part are assigned to that particular node.

2.3.2 Replicated Schemes

Replication is the allocation of a single file of the database to multiple sites. If there is no replication the distributed database problem is significantly reduced in the area of concurrency control and synchronization, but the cost of doing a transaction may increase significantly when the transaction must access data from a number of sites. The opposite extreme is full redundancy where each file is present at every node. This method will optimize response time in the case of queries, but will severely impact update performance and cost as all updates must be propagated to every site. Partial redundancy is the balance between these two extremes. Redundancy is important in achieving much of the promise of distributed systems. Without redundancy reliability goals can only partially be met as the unavailability of even a single file may be seen as "total failure" to some applications. Without redundancy the choice of where to store data is an all or nothing situation. Finally, in reality updates tend to be small and simple where queries tend to be quite complex and involve large amounts of data [ROTH81]. In this type of situation redundancy would have clear benefit but again would increase the complexity of the system from a concurrency and synchronization point of view.

2.3.3 Summary

There are several generalizations that have been made concerning the file allocation and replication issues.

Champine reviews the problem in regard to the size of the file and the percentage of exception rate (or remote request). Figure 2.1 summarizes his view [CHAM81]:

Exception Rate	File Size	Distribution Scheme
—	small	replicate
small	large	partition
large	large	centralize

Distribution Scheme
Figure 2.1

His view simply states that if the file size is small, replicate the file and propagate updates. If the exception rate is small but the file size is large, the best solution is to partition the data in order to place the fragments where they are most frequently used. Finally, if the file size is large and exhibits a high exception rate it may be necessary to centralize the file. In other cases where the size of the file has not been considered, studies have indicated that the maximum number of copies of a file should be one unless the ratio of queries to updates is greater than or equal to 50% (ratio=.5) [MURO85], [CASE72]. Although these views are gross simplifications of the data distribution problem, they appeal to intuition when considering the trade-offs between update costs, communication costs and response time. In addition, they serve to highlight the importance of understanding an application's use of data. It should be clear from the discussion thus far that one cannot reasonably decide where to put data unless they know where it will be accessed from and how often.

In summary, the data distribution problem involves a clear understanding of an application and its use of data. Data allocation however is only one aspect in the design of a distributed processing system. The allocation of hardware and software must be considered in the overall design, along with the allocation of data. The number of design factors increases substantially as decisions must

include such things as network topology, channel bandwidths, number of processors, storage capacities, program locations as well as data locations [HEVN84]. All of these decisions are highly interdependent and involve trade-offs in the areas of reliability, performance, development complexity (synchronization and recovery), growth and overall system cost [MARI84]. Due to the complexity of modeling all of these factors concurrently, most research has considered only individual design problems or the combination of only a few [HEVN84].

The remainder of this chapter will first review the research done in the area of "pure" file allocation and then survey extensions to these earlier models, which take into account hardware considerations and file interdependencies.

2.4 File Allocation Solutions

The first attempts to deal with the problem of optimal file placement used mathematical programming techniques. Nearly all of these models were linear integer programming problems to find optimal solutions. Most of these models were driven by system requirements as to performance, cost, minimum access delays, data management overhead and storage. Specific data models were assumed, with variants applied to different system resources. Generally however, these models worked under the following requirements [CHAM81]:

> Given: A description of user demand for service stated as volume of requests from each node of a network to each file
>
> Given: A description of resources available, such as network topology, link capacity, cost of storage, communications cost, etc.
>
> Determine: An assignment of files to nodes which minimizes total costs.

2.4.1 'Pure' File Allocation Models

The simpliest file allocation model was introduced by Casey [CASE72]. His model looks for the optimal node assignment of a single file which minimizes the total communications cost, under the assumption of a fully connected network with no response time or memory restrictions imposed.

One important contribution of his model is the distinction made between update and query requests. In the case of queries it is assumed that the copy of the file which minimizes total communication costs is chosen, whereas updates are propagated to all file copies. The model represents the total communication cost as a sum of the cost over individual nodes that result from a given file allocation. Casey's cost equation is paraphrased below [CASE72]:

$$C(I) = \sum_{j \in I} \left[\sum_{k \in \Sigma} (UT)_j (UC)_{jk} + QT_j \min QC_{jk} \right] + \sum_{k \in \Sigma} STR_k$$

where:

I	= index set of system nodes
j	= index for nodes
k	= index for file locations
STR	= fixed cost of storage for locating files at kth node
QT	= query traffic (emanating from node j)
UT	= update traffic (emanating from node j)
QC	= cost of unit of communication from node j to k for query
UC	= cost of unit of communication from node j

His model demonstrated that query costs decreased as the number of copies of the file were increased; however, a penalty is paid for storage and update costs. Rephrased, if storage costs were low and there were no updates, complete duplication would be cost effective. If storage costs were high and update activity was high, then one copy of the file would be optimal. Casey analyzes this trade-off by examining the cost function as the number of file copies is increased. This is done using a directed graph (referenced as "cost graph") where each vertex is a file assignment and has an associated value from the cost function. The edges of the graph are paths corresponding to the addition of a single file. Casey demonstrates the monotonicity of this graph, implying that it is "sufficient to follow every path of the cost graph until the cost increases, and no more" [CASE72]. This bounds the number of computations in both breadth and depth (commonly referred to as a Branch and Bound Search). Therefore, only a subset of the original tree (of 2 to the power of "n" nodes) needs to be tested. Even with this property, Casey's model

was proven by Eswaran to be NP-complete suggesting that heuristics were needed to efficiently deal with the problem [HEVN84].

Another approach to the problem was introduced by Chu. His model sought to minimize overall operating costs by determining the optimal placement of files under the constraints of response time and storage capacity. No distinction was made between query and update requests [CHU69]. This model again introduced a very large number of variables for even small problems, making it very costly, and in large problems computationally infeasible [CERI83].

Several comprehensive reviews of other models which address the pure file allocation problem can be found in [CERI83a], [HEVN84], [LEVI79]. Most of the proposed solutions apply different heuristic measures to reduce the computational complexity of the models. In general all of the models in this category assume a completely connected network topology and a completely defined distribution scheme in terms of storage capacity, storage costs, communication costs, frequencies, and user requests [HEVN84]. With all of these parameters well defined they attempt to obtain an optimal design for a very specific problem, namely the allocation of a single file. One model which was introduced by Morgan and Levin deserves special note, as it points out another weakness in the models discussed so far.

Morgan and Levin's model distinguishes itself from others by considering both program and data allocation. Their work points out the importance of considering the dependencies between programs and data in a heterogeneous environment. The dependency points out that while data can move easily from node to node, programs cannot, as in a heterogeneous environment different hardware and system software will exist. In this environment, program execution is limited to certain nodes. This is important as a transaction at one node may invoke a program at another node, which in turn needs data from a third node. Their model considers this restriction, by analyzing the optimal placement of both data and programs. In addition their model also

considers dynamic behavior and uncertain demand. Most of the previous models assumed "static" behavior, meaning that once the frequency of requests was determined they would not change. Also assumed was the complete availability and accuracy of access patterns. Morgan and Levin's model deals with dynamic behavior and incomplete information by minimizing costs over a number of different time periods during which allocation may change [LEVI79]. The practicality of changing the distribution over time has been debated however. As pointed out by [GROS80], once a distribution scheme is set up it remains fairly static in practice as redistribution could involve a tremendous amount of effort.

Ceri, et. al. and Hevner review similar work done by Fisher and Hochbaum [CERI83a],[HEVN84]. This model is an optimization algorithm for placing multiple copies of programs and databases over a network. They improve the work done by Morgan and Levin by developing several heuristics to generate feasible solutions to the problem and report practical experience.

2.4.2 File and Hardware Allocation Models

Various extensions have been made to the file allocation models discussed above, which explore different sets of assumptions and problems. These approaches relaxed some of the restrictions of other models by addressing such things as channel capacities and network topologies.

Mahmoud and Riordan consider the combined problem of optimal file allocation and channel capacity determination given a fixed network topology. The objective of the model is to minimize communication cost and storage cost subject to network delay and average file availability. The model was a nonlinear integer programming problem making it quite expensive; thus, an efficient heuristic was developed [MAHM76]. Their results have produced reasonably good allocation solutions [HEVN84].

Irani combines the file allocation, network topology and channel capacity allocation into a single problem. The model minimizes the total cost of file storage and communication capacities over different channels. The constraints imposed on the model include a maximum communication delay, minimum availability of single files and a minimum level of network reliability [IRAN79].

As a final example, Hevner reviews the work done by Casey [HEVN84]. Casey extended his original model by including optimal selections of network topology and channel capacities. Due to the size of the problem the topology considered is restricted to tree networks. Again, Casey developed heuristic techniques to solve the problem as the original model was nonlinear and contained integer and continuous variables.

2.4.3 Summary

In summary, earlier models developed in this category focused solely on the file allocation problem, leaving all other factors invariant. Extensions were made to these models to deal with program allocation and hardware allocation design choices. Several generalizations can be applied to these models. Most of these solutions were integer linear programming problems (i.e., each design parameter had a 0 or 1 allocation). These were far too costly, and hence infeasible to run for systems of any real size. Heuristic techniques were therefore developed to deal with the problems. These heuristics sacrifice optimality for practicality of use. Even with the use of these heuristics, the complexity of the problem has limited most research to dealing with the optimization of at most two resources at once [HEVN84].

Other major drawbacks of these models include their assumptions in regard to file usage and partitioning. These models do not address interdependencies between files which appear in realistic databases. They assume access of a given file from a given node, and do not reflect the demand for data access involving more than one file. As pointed out by [ROTH81], consider a join where each of two nodes has a file used in the new relation. A join request involving these

two relations could involve substantial communication costs. A file allocation scheme that places both relations involved in this query at a single site may be far more advantageous than if distributed (Although true in principal, this view may be slightly exaggerated as the amount of data involved could be reduced by a query optimization technique that utilizes semi-joins [HEVN84]). Finally, these models assume that complete files should be the unit of assignment of data to nodes. There is no consideration given to partitioning the files in order to reduce access and storage costs. While these models are important, they often obtain the optimal design for a very specific problem. The use of these models may be highly advantageous after the partitioning problem has been addressed and the physical distributed system is designed.

2.5 File Dependency Methodologies

As stated above, one of the major problems with the pure file allocation models is the assumption of a single file being the unit of distribution with no consideration given to file dependencies. Research in this area proposes methodologies and solution methods that consider the entire database schema as opposed to individual files. Some of the research presents guidelines and classification schemes to be followed during logical distribution design (non-automated). A notable contribution in this area is the work done by Baker [BAKE]. His methodology points out the major issues that must be addressed during logical design. Other contributions propose theoretical models and heuristic algorithms to determine the logical distribution. Examples from each category are reviewed below.

2.5.1 Baker's Model

Baker proposes a methodology in which logical distribution is defined as a "partition of a collection of related applications and their data into a maximum number of groups that have a specified low level of interdependence". 'Internodal dependencies' arise when an application transaction (or program) requires data from a remote node. The goal then is to minimize the

internodal dependencies and move toward 'nodal autonomy' or 'nodal interdependency' [BAKE].

In Baker's model, transaction or intergroup dependencies have an orientation, meaning an ownership of the transaction. Several quantitative measures are outlined that can be used in different combinations to classify intergroup dependencies. These include frequency of use, pattern of usage, required currency, level of consistency and timing required and the degree of remote data needed. In addition other classifications can be used to rate the dependencies, such as read versus update requests (where read would have a weaker dependency) or deferrability of the transaction.

Baker's approach to the design is iterative in nature and follows through six steps. These steps and their associated activities can be summarized as follows:

1. *Data Gathering:* Determine the number and types of databases, database structures, relations among the databases and information regarding the application make-up (i.e., split geographically or by function).
2. *Define Application Groupings:* Define application group structure such as order entry, production planning, etc.
3. *Assign Applications to Groupings:* Here a complete application which consists of a set of application programs is assigned to one group only.
4. *Assign Databases to Application Groupings:* Based upon knowledge of the applications and the data they require, assign databases to each application group (i.e., applications that make the most updates to data).
5. *Assign Transactions to Databases:* Here step 4 is ignored, and each transaction is assigned to the group that contains the data that are most closely related to the program or transaction. This is included to overcome the possibility of being assigned to an application group where the data the program most frequently uses is in another group.
6. *Analyze Dependencies and Evaluate Distribution:* The objectives of this step are to minimize the communication traffic and minimize the amount of data that must be copied between nodes. This step will be further described below.

In analyzing dependencies and evaluating the distribution (step 6), the dependency of each pair of groups is calculated from the set of transaction dependencies that occur between groups. These dependencies are characterized on a single transaction type by a) an orientation which relates the local to remote group; (b) the active component, or number which gives the frequency of use of

the transaction per day and (c) the passive component, or number which represents the number of bytes of remote data accessed by the transaction. When dependencies between groups are due to more than one transaction type, the dependencies are combined by adding the active components and measuring the union of passive components. Thresholds are then established judgementally to allow transactions to be categorized into four groups: (HH) meaning it is used frequently and accesses a large quantity of data, (HL), (LH) and (LL).

One constraint placed in Baker's model is that a logical distribution may not contain any HH dependencies. The dependency between two groups is then described using an orientation and three consolidated dependencies (HL, LH and LL). Values of these thresholds determine the number of groups in the logical distribution, the extent of the group's autonomy and the strength of the dependencies. Baker continues to discuss two types of dependency support, namely data communications when data currency and integrity are important or data duplication which yields good response time, management control and system availability.

Although Baker's approach may take several iterations to reach a satisfactory distribution, it highlights the importance of placing the data processing functions and associated data close to their users. Only in this way can a successful degree of nodal autonomy or an acceptably low level of interdependency be obtained, allowing the benefits of distributed processing to be utilized.

2.5.2 Theoretical Models

In contrast to Baker's trial and error approach, some research proposes theoretical methods to obtain logical distributions. Similar to the data allocation models, these approaches generally require heuristic techniques to make their use practical [NAVA84]. Many of these models are concerned with affinity among attributes and attribute clustering [CERI85],[NAVA84]. The work done by Navathe, Ceri , Wiederhold and Dou will be reviewed in some detail, so that an understanding of the clustering techniques can be achieved.

Navathe et.al. propose a set of algorithms to deal with the vertical partitioning problem (VPP). The approach consists of two phases where in the first phase the design is independent of specific cost information. The second phase performs cost optimization from knowledge of a specific application environment. The model also deals with three environments for the vertical partitioning problem; a single site with one memory level, a single site with multiple memory levels and multiple sites [NAVA84]. The multiple site allocation is most relevant to this review.

The inputs to the model are the logical accesses of the transactions to the attributes (i.e., number of accesses to object instances for one occurrence of a transaction at a site) and the relevant design parameters such as cost of storage, access and transmission. The steps in the first phase are summarized below [NAVA84]:

1. *Construct Attribute Affinity Matrix (AA Matrix):* The objective of this function is to construct an AA matrix which records the affinity or imaginary bond between attributes. The affinity measure is based on the logical access information which has been obtained. This includes whether or not a transaction uses a particular attribute, whether the transaction is retrieval or update and the number of accesses to the object for one occurrence of the transaction. The affinity measure recorded is the sum of these accesses per time period (i.e., per day).
2. *Cluster the Attributes:* The objective of this function is to group the AA matrix so that attributes with high affinity are clustered together as are attributes with low affinity. This is accomplished through a heuristic algorithm that diagonalizes the AA matrix to produce blocks of jointly accessed data items.
3. *Partitioning:* The authors provide two mechanisms for partitioning. The first provides partitioning for non-overlapping fragments. This attempts to find the ideal location on the matrix to form the partition of two non-overlapping fragments, so that the fragments are balanced with respect to transaction load. In general terms, if 'n' is the number of attributes, 'n-1' points on the diagonal are considered. At each point, the matrix is "split" into an Upper and Lower fragment. A count is made of the total number of access of transactions that need only fragments in the current upper block (labeled (CU)) and current lower block (labeled (CL)), and a count of the number of transactions that need both fragments (CI). The goal then is to select the point in the matrix such that the goal function (z) is maximized: max $z = CLxCU - CI^2$

   The second mechanism allows for partitioning with overlapping fragments. This requires the use of two points on the diagonal, x1 and x2 where attributes between x1 and x2 constitute the intersection. The goal function is the same for the non-overlapping case however consideration is given to read only versus update requests. In the case of read only there are advantages to sharing the data among the two fragments whereas updates need to be directed to both fragments for consistency.

The process as presented is iterative, as it is most likely that the vertical partitioning will result in several fragments. In order to reduce the computational complexity of the problem, the authors chose a suboptimal approach whereby each application of the VPP produces two fragments. These fragments then become independent subproblems or fragments on which the algorithms are repeated to further split the fragments. This is repeated until no further benefits are gained [NAVA84].

The final phase of the methodology deals with the allocation of these fragments to sites. Four cost factors are considered in this stage. These include the cost of irrelevant attributes accessed within a fragment, cost of accessing fragments for retrieval and update, storage cost and transmission cost. These cost factors are then assigned weights according to their importance in the overall optimization model. In a distributed environment the transmission costs receive the highest weight in their model. To summarize, a table of partitions versus allocations is maintained and for each possible partition the algorithm attempts all possible fragment allocations (m2 cases for m sites). The "least cost" pair is selected. In the case of replicated sites, a final algorithm is invoked. This algorithm looks at each fragment independently and allocates additional copies to sites until no further benefit is gained [NAVA84].

Other optimization models exist which address horizontal and replication issues [CERI83a,CERI83b]. These models analyze the logical distribution in terms of objects and links and the relationships between them. They assume explicit knowledge by the user in terms of potential uses of the database, transaction frequencies and cardinalities (number of instances) of objects and links, etc. Given these inputs they produce formal solutions to the design problem.

2.5.3 Summary

The methodologies and solution methods which have just been reviewed address the distribution problem from a higher level than the pure file allocation models. This is done by assessing the

dependencies between transactions or the affinity or bonding between attributes. These approaches highlight the many issues that must be taken into account during logical design. It is important to note that there does not appear to be any one model that addresses horizontal and vertical partitioning under one methodology, due to the complexity involved.

2.6 Related Work

As a final note, Hevner has pointed out the importance of combined research in the areas of data allocation and query optimization. As he states, the data allocation problem generally assumes a given data access pattern and the query optimization problem assumes a fixed data allocation. If these assumptions were generalized, designers could develop models that combine the allocation of files and the support of query processing in the most efficient manner [HEVN84].

Hevner reviews some of the recent work done in this area. The work of Elam minimizes the amount of data sent for a specified set of query processing strategies, under the constraint that one file is stored separately from others in the query to promote parallel processing. Another approach by Apers develops an algorithm to minimize data transmission costs by clustering relations. The constraint placed on this model is that only one copy of each relation is stored in the system (i.e., no redundancy) [HEVN84]. In agreement with Hevner, further research in this area is needed.

2.7 Conclusions

This chapter enumerated the many issues associated with the distribution of data and file allocation in a distributed environment. Several attempts to deal with these problems have been addressed in the literature. Early models focused solely on the optimal placement of files, with extensions added later to address program and hardware allocation in parallel. The major weakness of these models was the assumption of a file being the unit of allocation with no incorporation of file partitioning or file interdependence. Other methodologies and solutions were

presented that address these issues. These methodologies focused on transaction dependencies and bonding between attributes when analyzing the placement of data.

In summary, in a distributed environment a database designer will be faced with many different problems, including how to partition files, where to place these partitions and what degree of redundancy should be incorporated. Although an optimal solution to all of these problems is not realistic at this time, various models exist for different classes of these decisions. Integration of a solution to some of these problems into an interactive tool, would greatly reduce the complexity involved. Such a tool would provide a building block on which the designer could utilize his judgement, to design a feasible distribution scheme.

CHAPTER 3

DESIGN

3.1 Objectives

The objective of this project is to design a tool that can be used to aid the database designer in the distribution of data. Total optimality and automation of all aspects of the distribution problem still requires a great deal of research. A tool therefore can only expect to provide guidance and recommend solutions, as ultimately the design process still relies to a large extent on the designer. In that regard, the distribution algorithms developed should be integrated into an interactive design tool. With such a tool, the designer can review the steps that have been taken and modify the results as desired.

3.1.1 Design Overview

The tool being designed is an extension to two automated tools in existence today. The first is the "document_handler" program which processes application documents and determines the functional dependencies (FD's) they represent. The FD's are used as input to the "bern2" program, which produces a database in third normal form (3NF). The data distribution tool (hereafter referred to as "dist_data") then deals with the question, what if I now want to distribute the data ? A high level overview of the functionality of these tools is required in order to understand the interdependencies between them.

3.1.1.1 Document_handler Program

In distributed processing, one of the fundamental keywords is *document*. A collection of the documents used in an organization will tell you a great deal about the database schema required. A complete set of user's documents should supply all of the data items used by an organization. Problems arise though, as different user documents refer to the same data item by different names (synonyms) or the same name is applied to different data items on different documents

(homonyms). Another problem arises when analyzing these documents, as some data items names appear on a user document but are not actually stored in the database. These are derived data items that may be computed from other data values. The database designer must examine these documents to generate a nonambiguous list of data items for the database [WOEL71]. This can be an arduous task.

The document handler tool was developed to provide automated assistance in interpreting these documents. The tool takes as input all user documents and their associated columns, along with a specification of their use as input, resident or output documents. The documents and columns are scanned one by one, and the following activities take place [COHE79]:

— Deletion of synonym/homonym names
— Removal of insignificant columns (some columns may not be included in the database due to their nature. A signature field for example)
— Solve undeclared output

These activities are accomplished by interactive dialogue with the user. The system contains a sophisticated mapping between all documents and columns. This allows a cross reference listing of any column to any document as well as a listing of any document, it's type and all columns it contains. This information is stored and maintained by the program.

Once a unified list of document names and attributes has been derived, keys are specified for each document. These keys are then used in conjunction with the columns in a document to form functional dependencies. These functional dependencies are then used as input into the bern2 program.

3.1.1.2 Bern2 Program

The bern2 program is a tool which automates the steps of Bernstein's algorithm, that produces a database schema which is in third normal form (3NF) with a minimal number of relations. A database in 3NF contains no extraneous attributes, no partial dependencies and no transitive

dependencies (a transitive dependency exists when you can get from a key to a non-key to a non-prime). The removal of extraneous attributes and redundant FD's is particularly applicable to the FD's introduced by the document_handler program, as many documents will use the same keys and contain a great deal of data items in common.

The steps of Bernstein's algorithm can be summarized as follows [COHE79]:

1. Eliminate from the functional dependencies those data items that can be derived from other functional dependencies (extraneous attributes)
2. Eliminate from the set those functional dependencies that can be derived from the remaining set of FD's (redundant FD's)
3. Group the remaining functional dependencies into sets with equivalent left hand sides
4. Merge the groups that have equivalent left hand sides (i.e., keys)
5. Remove transitive dependencies from the data items
6. Construct relations based on the groups of functional dependencies.

3.1.1.3 Data Distribution Tool

The dist_data tool is the focus of this project. It extends the work of the tools just reviewed, by addressing the fact that the resultant database may be used in a distributed environment. It utilizes much of the information obtained in the document_handler and bern2 programs. Specifically, the document_handler provides critical information about an application which is needed to make intelligent distribution decisions. This information includes application transactions in the form of input and output documents.

The bern2 program removes all extraneous attributes and transitive dependencies to produce a database in 3NF. These relations coupled with the information on customer documents provide the basis for defining meaningful partitions and file placement onto nodes in a network.

As previously discussed, there are two main concerns when dealing with data distribution. The first is to evaluate the utility of partitioning data objects into fragments, and the second is, once partitioned how these fragments are allocated to nodes on a network. This tool addresses both these issues. The intent is not to promise true optimality, but to provide a feasible solution to

these complex issues.

3.1.1.4 Limitations

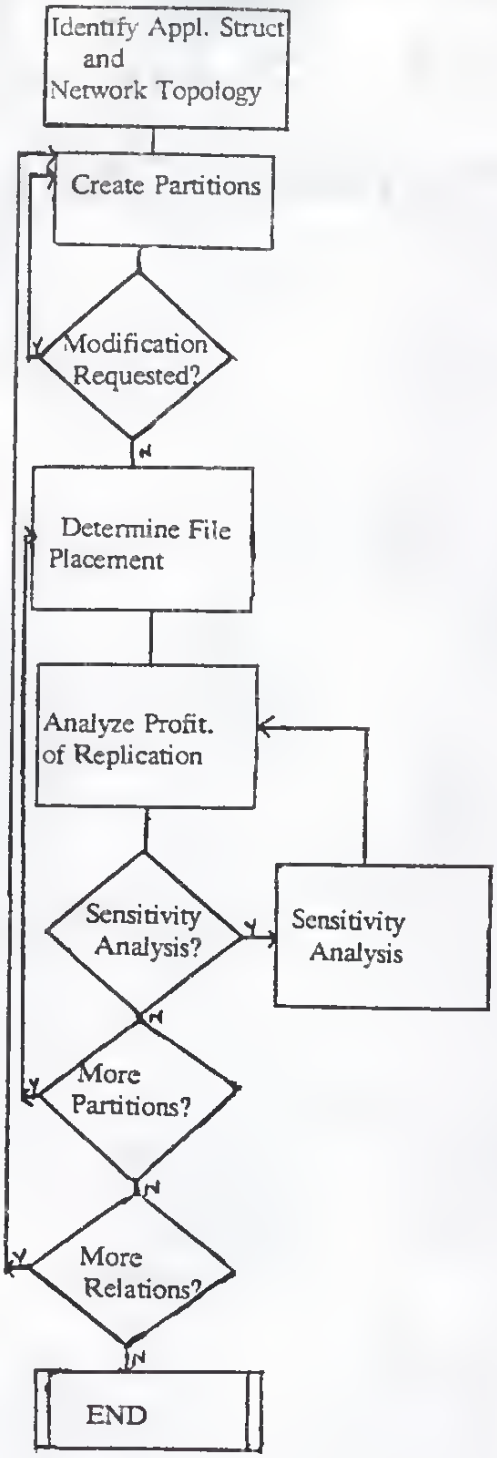
Some of the requirements originally outlined for an automated document handler [WOEL81], were not implemented in prototype document_handler program. Specifically, an indication of the frequency of use on a per document basis and the ownership of data items (i.e., what document(s) own each particular data item). These pieces of information are required in the distribution algorithms presented. As these modifications would not prove difficult to make, they will be assumed as input for the purpose of this project.

3.2 Detailed Design

This section presents the detailed design of the dist_data tool. The design algorithm involves four basic steps that can be summarized as follows:

1. Identify application structure and network topology
2. Create Partitions/Files
3. Determine File Placement
4. Analyze Profitability of Replication (Branch and Bound)
5. Sensitivity Analysis

The steps of the algorithm are iterative in nature; step 2 is repeated for all relations produced from bern2, and steps 3, 4 and 5 are repeated for each file or partition generated. Figure 3.1 provides a flowchart of this activity. The remainder of this chapter will discuss each step of the design algorithm in detail.



Flowchart of Dist_data Process

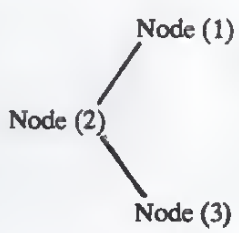
Figure 3.1

3.2.1 Step 1 : Identify Application Structure and Network Topology

Data distribution cannot begin without understanding the organizational structure and distribution scheme for any particular application. The first step in the design process is to gather this information from the user. For ease of use the tool will prompt the user for the needed information, which includes:

1. Number of nodes on the network
2. Identification of organizational groupings (i.e., INVENTORY, BILLING, etc.)
3. On a per organization basis:
   - — Which node the organization resides on
   - — A list of nodes which interconnect with this organization and the associated communication costs of each interconnection
   - — A list of the documents used by the organization

With this information in hand, several required pieces of information can be compiled. First, the application network topology has been defined. The system may now build cost tables which reflect the transmission costs incurred for queries and updates. It is important to point out one requirement of the system which is needed by later algorithms. The file placement algorithm (step 3) and replication algorithm (step 4) require cost figures to be associated between any two pairs of nodes on the network (i.e., a fully connected network). The problem can be demonstrated as follows. Consider a system with three nodes, where nodes 1 and 3 are connected to node 2, but no connection exists between nodes 1 and 3. This topology is illustrated in figure 3.2.



Example Network Topology
Figure 3.2

The cost tables which need to be built require that a cost be associated with nodes 1 and 3. In the example above, this can be accomplished by taking the sum of the communication costs between nodes 1 and 2, and nodes 2 and 3. In a situation where multiple paths are available, the least cost path should be associated with the "imaginary" link. The system should detect these missing links and compute the cost factor. Once a cost is associated with all of the nodes in the network, the cost table can be built. Figure 3.3 shows an example of a cost table for three nodes. Reading the table across tells you that the cost of communication between node 1 and itself is zero, node 1 and 2 is 8 and between nodes 1 and 3 is 12. This information is required in steps 3, 4 and 5 of the algorithm.

(Nodes)

	1	2	3
1	0	8	12
2	8	0	20
3	12	20	0

Network Cost Table
Figure 3.3

The other piece of information obtained from the user deals with the application structure. The system now knows on which nodes each document or transaction resides. This node identifier must be stored in conjunction with the document entries as it is vital in determining a meaningful partitioning of relations (this will become clear after reviewing the algorithm for step 2).

The information obtained in this step serves as a framework for all other steps. It should be noted that this is the only step which *requires* information from the user (all other steps of the algorithm make use of the information provided by the document_handler and bern2 programs). Appropriate error detection and recovery will be provided when parsing the user input. This includes syntactic checks as well as checks for invalid document names or invalid interconnect nodes.

3.2.2 Step 2: Create Partitions/Files

Step 2 of the algorithm is a key step in the distributed design process. In this step, the relations produced by the bern2 algorithm are analyzed to determine if partitioning should be applied. Recall that initially the application has a list of documents which they own and execute. The document_handler program removes all homonyms, synonyms and insignificant columns to insure that the data in the document is interpreted correctly. The bern2 program then takes these documents (represented as FD's) and produces a 3NF database. These relations do not reflect customer usage however, as one relation may now contain pieces of data from several different documents. Chapter two reviewed the importance of analyzing the application's use of data , particularly in regard to the dependency which exists between files. These dependencies may reveal clusters of data which belong together or that could form meaningful partitions. Once these partitions are identified, the file allocation algorithms can deal with the physical placement of these fragments onto sites in a network.

Several different approaches which have been taken in dealing with partitioning were discussed in Chapter two. The heuristic chosen for this design process, is to analyze the frequency of use of each attribute in the relation on a per node basis, and cluster these attributes together into a file. This approach, although simplified is very similar to the approach used by Baker. The justification for this algorithm is as follows:

- The grouping of attributes by frequency of use agrees with the underlying message in the literature; store the data where it is most frequently used.
- Partitioning the data in this manner usually results in partitions being stored at the same node as the owner of those attributes. This proves beneficial, as most of the time people are "greedy" in the sense that they want to maintain control over the data they own.
- Availability of information: The document_handler program contains all the needed information to analyze the relations in this manner. Each document entry contains all related data items and the frequency of use. Step 1 of the distribution algorithm also associated a node with each document. All of this information is readily available and feasible for the scope of this project.

In order to accomplish the partitioning, three steps are needed. These steps build a document table associated with the relation and sort it by node, analyze the data frequencies and then form partitions based on this frequency. At the conclusion of this step the designer may review the partitioning which has taken place, and make modifications if desired. These functions work on one relation from the bem2 algorithm at a time. They are therefore repeated until all relations have been processed.

The phases of this step are reviewed in detail below.

3.2.2.1 Build Document Table

Figure 3.4 and 3.5 contain one relation produced by bem2 and three document entries as they would look after step 1 of the distribution process.

REL1 (KEY_ATTR, ATTR1, ATTR3, ATTR5)

Bem2 Relation
Figure 3.4

NODE	FORM	KEY_ATTR	ATTR1	ATTR2	ATTR3	ATTR4
1	form1	10	10*	10	10	10*

NODE	FORM	KEY_ATTR	ATTR5	ATTR8
3	form2	10	10	10

NODE	FORM	KEY_ATTR	ATTR3	ATTR8
3	form3	5	5	5

Document Entries
Figure 3.5

The first step is to build a document table which corresponds to the relation being examined. In this example, only those documents using the key attribute and attributes 1,3 and 5 are of

concern. This table is then sorted by node for ease of processing in the next step. Figure 3.6 shows the document table at the conclusion of this step. Note that attributes 2,4 and 8 used by the forms, do not appear in this table.

NODE	DOC	KEY_ATTR	ATTR1	ATTR3	ATTR5	ATTR7
1	form1	10	10*	10		
3	form2	10			10	
3	form3	5		5		

Document Table
Figure 3.6

3.2.2.2 Analyze Frequency of Use

In this step, each data item or attribute is examined individually and an aggregate usage at each node is determined. Two important assumptions are included in this step:

1. Keys are not analyzed, as the key to this relation is required in any partition created (i.e., the key must be duplicated for access)
2. An access to an attribute by the owner is taken as an update transaction (recall that an owner(s) of a data item must be indicated. This is represented by a '*' in these examples. It is possible for more than one owner to exist for any given attribute.) These accesses represent exception rates such as new customers being entered, etc.

Using the example in Figure 3.6, the following data would be compiled:

- ATTR1: 10 updates node 1
- ATTR3: 10 queries node 1, 5 queries node 3
- ATTR5: 10 queries node 3

3.2.2.3 Create Partitions

This step processes the frequencies above to determine the partitioning. In most cases the aggregate total of query and update requests from a node is used to determine the owner. In the case where a query count from one node equals the update count from another node, the update access is given the higher priority (It is recognized that further heuristics could be applied when

queries are very close to updates, etc. Further heuristics are not addressed in this design). Each attribute is therefore associated with the node which contains the highest frequency of use. The final step combines all attributes from the same node into one file.

The final partitions resulting from the example above are:

FILE1 = (KEY_ATTR, ATTR1, ATTR3)
FILE2 = (KEY_ATTR, ATTR5)

3.2.2.4 Query User

After the partitions have been formed, the results should be displayed to the user. At this point the user will be allowed to modify the partitioning if desired. Three operations will be allowed:

1. MOVE ATTR ___ to FILE ____
2. CREATE FILE ____ (This will create a new partition, which may then be populated through a series of MOVE requests)
3. MERGE FILE ____ and FILE _____

These operations will again be prompted for, so that minimal effort is required on the user's behalf.

3.2.3 Step 3: Determine File Placement

Once the relation has been partitioned into files, each of these files needs to be analyzed individually to determine where on the network they should be placed. The objective function chosen is to place the file at the location which minimizes the overall communication costs in regard to update and query requests. To accomplish this, transaction tables must be built to indicate the frequency of requests issued against this file. Query and update tables are then built and analyzed to determine the optimal placement of the file.

3.2.3.1 Build Transaction Table

A transaction table needs to be built for each partition or file produced in step 2. The table contains a count of the number of queries and updates issued against this file from each node.

This information was previously gathered for all attributes. This step however is only concerned with those attributes associated with the partition being analyzed. The same rules for distinguishing between query and update requests in step 2 , also apply to this step. In addition, one other assumption is made:

— It is assumed that all attributes in a document/file are accessed together (i.e., cannot access individual attributes out of a file, the entire file is retrieved). This implies that if a form has some queries and some updates against the relation, an update transaction is assumed. It would not seem appropriate to consider these as two separate accesses, so one access is assumed with updates weighing more heavily.

The output of this process will be a transaction table which holds the associated access frequencies.

Figure 3.7 shows an example of a transaction table (an additional node has been added from previous examples).

Nodes	Query	Update
1	10	10
2	30	10
3	15	0

Transaction Table
Figure 3.7

3.2.3.2 Build Cost Table

Cost tables must now be created to reflect the query and update costs that would result from placing the file on any given node. These costs are computed using the transaction table and the network cost obtained in step 1.

For example, assume the following entries exist in the transaction table and network cost table for Node 1:

Transaction Table

Node	Query	Update
1	10	10
2	30	10
3	15	0

Network Cost

	1	2	3
1	0	8	12
:			
:			

The query costs associated with placing the file at node 1 can be computed as follows:

1. Node 1 to itself: cost = 0
2. Node 2 to node 1: cost = 240 (30 query requests from node 2 at a cost factor of 8)
3. Node 3 to node 1: cost = 180 (15 query requests from node 3 at a cost factor of 12)
4. Total query costs incurred = 420

The same algorithm is used to determine update costs. The total cost associated with each node is the sum of query and update costs. These costs are computed for every node on the network. The node representing the minimum cost is chosen for file placement.

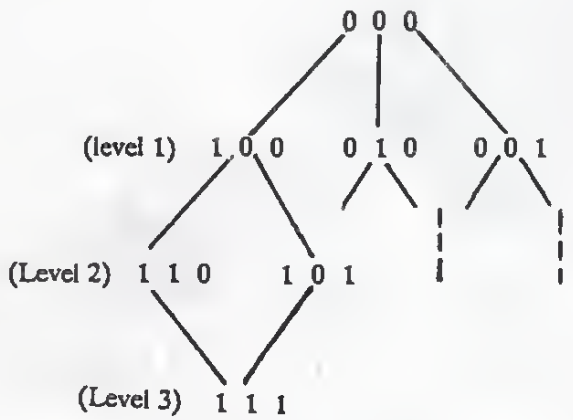
3.2.4 Step 4: Determine Profitability of Replication

The next step of the distribution is to determine the profitability of replication. This is accomplished by building a decision tree and performing a branch and bound search.

In theory, the branch and bound search looks for an optimal solution by defining initial upper and lower values of the objective function (in this case the objective function is the minimization of communication costs). From the feasible solutions, the best solution is made the upper value (U) of the problem. All other solutions are matched against this solution in an attempt to find a better solution. Any solution which produces a value higher than (U) are deleted, as further branching would not lead to a better solution. This process continues through a series of iterations in an attempt to find the optimal solution [GREEN78].

The branch and bound search technique was reviewed briefly in Chapter two. This discussion highlighted the fact that even though this search procedure bounds the number of required computations, heuristics are still needed to reduce the potential computational requirements. In this light, the hueristic identified for this design is a simple "greedy" heuristic. At any given point in the search, only the best solution is kept. Although it is realized that this can not guarantee the true optimal placement of the file, it provides a reasonable placement within the scope of this project.

To correlate the above discussion to the file allocation problem, consider the placement of a file on a network consisting of three nodes. Three levels of the decision tree are depicted in figure 3.8 for purposes of this discussion.



Partial Decision Tree
Figure 3.8

At the first level, each vertex represents a file assignment to a given node (denoted by 1's in those positions corresponding to file nodes, 0's elsewhere). The file placement algorithm discussed in step 4, determines the optimal placement of one copy of the file to the network. For this example, assume this is node 1 (represented by a 100 at level 1 on the graph). With the "greedy"

hueristic applied, the search will now only be concerned with replication schemes that involve node 1. The search proceeds to level 2, where the replication choices are a copy of the file at nodes 1 and 2, or a copy at nodes 1 and 3. The cost associated with each of these choices is analyzed. Obtaining the cost figure is very similar to the file placement algorithm; however, it is modified slightly to account for the replication. For example, when analyzing the costs associated with file copies allocated to nodes 1 and 2 the following changes will be made. The cost associated for queries at both node 1 and node 2 are zero, as each node will now have a local copy. A query from node 3 can now be satisfied from either node 1 or 2, so the lower cost is assumed. Update costs are increased however, as updates must be propagated to all file copies. In this case, the update cost associated with the replication would be the sum of update costs associated with nodes 1 and 2. Again, the total cost is the sum of query and update costs. If this cost is less than the cost associated with only 1 copy of the file, then this choice is now considered the "best" placement and the search continues to level 3 (which represents total replication in this example). If the costs associated with replication is higher, the search has ended (after both nodes are tested).

3.2.5 Step 5: Sensitivity Analysis

As the replication decision may be far from optimal, a sensitivity analysis is being provided. The intent of this step is to allow the designer to modify the communication costs associated with a given topology and re-analyze the file placement. In other words, the designer will be able to request a reiteration of the branch and bound search with the communication cost increased by some specified amount. If the result after this run differs significantly from the first, this may indicate that replication is desirable in either case. For example, if a 5% increase in communication costs shows that replication is desirable this would indicate that the file placement is very sensitive. This type of sensitivity may imply that replication is in order.

At the conclusion of this step, the file allocation process has been completed for one file or partition. Each original relation may be split into several partitions. Steps 3, 4 and 5 (if desired) will need to be repeated for each partition.

3.3 Summary

This chapter presented the design for the dist_data tool. This tool takes the information regarding customer documents coupled the 3NF relations they represent, and automates a distribution analysis. This analysis includes recommended partitioning for the relations based on file dependencies and frequency of use, and the allocation of these partitions onto the nodes in a network. The allocation scheme is based on minimizing communications costs for update and query transactions. The process is iterative, examining one relation at a time and then each resulting partition in that relation. This process is repeated until all relations have been analyzed.

The interface to the tool is interactive in nature, allowing the designer/user to make modifications if desired to appropriately steer the distribution design. A minimum amount of error detection and recovery has been provided.

Chapter four contains a detailed example of the dist_data operation.

CHAPTER 4

EXAMPLE DESIGN

4.1 Example Design/Results

This chapter presents a complete design, following the steps outlined in Chapter three. The system used in this example, models a manufacturing firm which consists of three organizations: CUST_REL, ORDER/BILLING and INVENTORY. These organizations will be distributed across a three node network.

The example starts by showing a logical listing of the documents, as they would appear at the end of part one of the document_handler program. Appendix I contains a list of these documents. Each document is labeled with a type (i.e., INPUT, RESIDENT or OUTPUT) and has a document key specified. An owner is also indicated for each data element (denoted by a '*').

Appendix II contains the FD's as produced by the document_handler. These FD's are used as input into the bern2 program. At the conclusion of the bern2 run, two 3NF relations are produced.

The dist_data tool commences after the results from the above programs have been obtained. Appendix III contains a step-by-step example of the procedures used. The remainder of this chapter will summarize the processing that occurs at each step.

The first step of the dist_data tool, is to identify the application structure and network topology ((III.1). This notation will be used throughout the chapter for Appendix III, point 1). Information is solicited from the user regarding the organizational structure, the documents used and the network topology. Note that in the example, the system must detect the missing node interconnection between nodes 2 and 3, and search for the least cost connection (in this case there is only 1 path between nodes 2 and 3). At the conclusion of this step, a network cost table has

been built and each node has a list of associated documents that are resident at the node.

The next operation examines the utility of partitioning (III.2). This step of the algorithm deals with one relation at a time, which in this case is REL1 produced by bern2. The table is constructed by identifying all documents which use the attributes associated with this particular relation. The owner of a data item is indicated by a '*', and the numbers in the table represent the frequency of use. These frequencies are then analyzed on a per data item basis to determine the owner (i.e., node) of that data item (denoted by an 'X'). All data items with a common owner are then merged to form a partition. In this example, Node 1 exhibited the highest frequency of use for all data items and therefore no partitioning was done. At this point the designer may display the partitioning that has been created and perform modifications if desired (see section 3.2.2.4).

Step 3 of the algorithm determines the placement of all partitions/files produced. It deals with one partition at a time, so it is repeated for each partition produced in step 2 (in the example so far, only 1 partition has been created). Step 3 begins by building a transaction table (III.3). This table represents the aggregate usage of this file from all nodes. Recall from section 3.2.3.1, there are two critical assumptions used when building this table. The first is that access by an owner is interpreted as an update transaction. Secondly, a document or file is assumed to be accessed in entirety. The importance of these assumptions can be highlighted by examining the HIST_FILE document. This document accesses c_no and part_no as a query request and tot_price as an update. This transaction from node 1 is interpreted as an update request of frequency 30. The remainder of the requests from node 1 are query only, so the table is populated with 30 query requests and 30 updates.

Once the transaction table has been built, the cost for query and update resulting from placing the file on any given node, can be determined. The network cost table (from step 1) is used in

combination with the transaction table, to compute these costs. For example, Node 1 issues 30 queries and 30 updates. If this file were placed on Node 1, there would be no cost for queries or updates from Node 1, as all the information is local. However, there are query and update requests to this file from Nodes 2 and 3. Node 2 issues 6 queries and 5 updates with a cost factor of 8. Reading the query, update and network tables down the Node 1 column, this means if the file were placed at Node 1, a cost of 48 would be incurred in queries and 40 in updates from Node 2. The computation continues in this manner in order to analyze the total cost from any node. The node which represents the minimum cost is recommended for the file placement (Node 1, in this example).

After a single file placement is recommended, the tool looks at the profitability of replication (III.4). This represents step 4 of the distribution algorithm. As previously discussed in section 3.2.4, the tool progresses only with the "best choice" at any given time. From the preceding step, node 1 is chosen for the file placement. Replication options at level two of the cost graph include two copies of the file, at either nodes 1 and 2, or node 1 and 3. The costs are recalculated under the assumption of replication, to determine if either option results in a lower cost. In the example, replication proves profitable at nodes 1 and 3. Therefore, the final recommendation for file 1 is to replicate the file at nodes 1 and 3, at a total cost of 572.

The final step of the algorithm (step 5) allows the designer to perform sensitivity analysis (III.5). This has not been illustrated in the example, but would involve receiving a cost increase parameter from the user (i.e., 5%), and recomputing the cost tables and reanalyzing the distribution.

Appendix III.6 displays the network after the placement of file 1. The communication traffic resulting from this placement is also illustrated. In analyzing this placement, there are two points of interest. A general view taken by [ROTH81] discussed in Chapter two, stated that in reality queries tend to be large and complex whereas updates tend to be small and simple. A file

placement that minimizes query traffic is therefore beneficial. The file placement produced in this algorithm has modeled this assumption. The major query requests have been fulfilled at both nodes 1 and 3. Updates do incur communication expense but these updates are small (part_no and tot_price). The frequency of access to this file at Node 2 is so low, that further replication would not prove advantageous. The second theory discussed in the literature, is that a single copy of a file suffices under most conditions, if the ratio of query to updates is more than or equal to 50% [CASE72]. Looking back at the transaction table for file 1, the ratio of query to update (77 queries, 37 updates) is 48%. The results of this algorithm have stayed within these guidelines (it should be noted, that this comparison holds throughout the remainder of this example. All file placements meet these guidelines).

Processing now returns to step 2 of the algorithm, to begin analysis of the next bern2 relation (III.7) At the conclusion of this step, REL2 has been partitioned into two files.

Step 3 of the algorithm: determine file placement, is once again invoked for File 2 (recall that this step analyzes a single partition at a time). The processing steps are identical to that of file 1; the node that minimizes the total costs is found (III.8) and then replication is analyzed (III.9). The final recommended placement for file 2, is again two copies of the file residing at nodes 1 and 3. The total cost for this placement is 264. III.11 shows the overall view of the network with both files 1 and 2 allocated.

To complete the processing, one partition is left to analyze (File 3). The algorithm returns to step 3, to determine the single file placement for this partitioning (III.12). Again, the processing is identical to that of file 1 and 2 and is illustrated in III.12 - III.14. In the case of file 3, replication is not profitable and therefore a single copy of the file is placed at node 3. The cost associated with this placement is 0.

Appendix III.15 shows the final results after the dist_data program has processed all relations and partitions. To summarize, file 1 and 2 are replicated on nodes 1 and 3, and a single copy of file 3 exists at node 3. The overall cost resulting from these allocations is 836.

CHAPTER 5

CONCLUSIONS

5.1 Discussion of Results

The goal of the system presented in this paper is to *assist* the database designer in finding solutions to the problems of file partitioning and file placement in a distributed system. The basic premise of the system is that user documents contain the needed information to make reasonable distribution decisions. A complete set of user documents supply the data items used by an organization and model the transactions (in the form of input and output documents). This information, coupled with an understanding of the ownership of these documents, allows file partitioning and file placement to be made based on the frequency of use and ownership of the data items. By automating the analysis of this data and proposing solutions, a great deal of complexity is removed from the designer. This is not to suggest that this is the optimal solution nor the only solution to the problem, as there are many issues which can impact these design choices. As such, the system was designed to be interactive in nature, allowing the designer to make modifications as required to appropriately steer the design process.

Research continues to address the key problems of file partitioning, file placement and redundancy considerations, in an effort to find optimal solutions. The underlying message in all these solutions is clear however; a major determinant in the increased use of distributed processing will be an increased ability to get data where it is needed, and used most frequently.

5.2 Extensions

Numerous enhancements could be made to increase the benefits derived from this system. These include extensions to the partitioning and file allocation heuristics, as well as the overall user interface to the system.

The partitioning scheme developed for this system does not directly address partitioning by occurrence, or horizontal partitioning. Although the system will handle the same organizations existing at multiple nodes and maintaining ownership of the same documents, it will assign the file to one of the two nodes only. For example, there is nothing to preclude the user from entering the ORDER/BILLING organization on two different nodes, each owning the same documents. If the frequency of use were the same against each document, the system will arbitrarily designate one node as owner. The single file allocation of this file would result in the node that represents the minimum communication cost. In this case the algorithm will most often conclude that replication is profitable at the second node, unless the communication costs vary significantly. The algorithm will not however, correctly address the other remote requests to this file, as it assumes both files are identical. The second problem arises when the frequency of access varies between the two locations (i.e., more activity with certain part numbers than others). In this case, the algorithm will cater to the node which exhibits the higher access rate. These problems point out the importance of allowing input from the designer, as the designer can easily remedy this situation by creating and allocating new partitions. A more sophisticated method of detecting and dealing with horizontal partitioning would be desirable however.

Another specific enhancement that could be applied to the partitioning algorithm, is to supply an improved heuristic for differentiating between updates and queries. The current design gives weight to an update in the case where query and update rates are identical, but does nothing more. It would be desirable to further analyze this difference and extend the heuristic (i.e., if there were only 10% more queries than updates, should the algorithm still favor updates ?).

Other limitations of the system exist in the file allocation and replication heuristics. The model is very restrictive, in that it only considers query and update communication costs. No consideration is given to such things as storage restrictions and costs, communication channel load, etc. Chapter

two highlighted several different models which take various system resources into consideration. The integration of any one (or more) of these issues into the model would certainly provide a more refined file allocation.

Finally, limited attention was given to the design of the user interface to the tool. A primitive method of displaying and modifying the information was described, that although is sufficient, could be greatly enhanced. One could envision a menu driven system with graphical display capabilities, in which the designer could control the total execution of the system, requesting any of the steps to be run in any order. The graphics would allow a cleaner display of the processing at any given step, or a visual view of the network topology and file placement.

[BAKE] Baker, Charles T., "Application Analysis for Nodal Autonomy", International Business Machines Corporation, Poughkeepsie, New York.

[BRAY81] Bray, Olen H. "Distributed Database Design Considerations", *IEEE Tutorial on Distributed Processing*, Third Edition, IEEE Computer Society Press, 1981 (ppgs 451-464)

[CASE72] Casey, R.G., "Allocation of Copies of a File in An Information Network", *AFIPS Conference Proceedings*, Vol. 40, Spring 1972 (ppgs 617-624)

[CERI83a] Ceri, Stefano, Navathe, Shamkant and Wiederhold, Gio "Distribution Design of Logical Database Schemas", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 4, 1983 (ppgs 487-503)

[CERI83b] Ceri, S. and Navathe, S.B., "A Methodology for the Distribution Design of Databases", *Compcon*, Spring 1983 (ppgs 426-431)

[CERI85] Ceri, S. and Pernice, B., "DATAID-D: Methodology for Distributed Database Design", *Computer-Aided Database Design: The DATAID Project*, A. Albano, V. DeAntonellis, and A. Pileva (editors), Elsevier Science Publishers B.V., 1985

[CHAM81] Champine, G.A., "Six Approaches to Distributed Databases", *IEEE Tutorial on Distributed Processing*, Third Edition, IEEE Computer Society Press, 1981 (ppgs 480-483)

[CHU69] Chu, Wesley W., "Optimal File Allocation in a Multiple Computer System", *IEEE Transactions on Computers*, Vol C-18, No. 10, Oct. 1969 (ppgs 885-889)

[COHE81] Cohen, Meir, "A System for Automatic Generation of Relational Databases", *A Master's Report*, Kansas State University, 1981

[DRAF80] Draffan, I W. and Poole, F., "The Classification of Distributed Database Management Systems", *Distributed Databases*, Cambridge University Press, 1980 (pgs 57-81)

[GAVI86] Gavish, Bezalel and Pirkul, Hasan, "Computer and Database Location in Distributed Computer Systems", *IEEE Transactions on Computers*, Vol. c-35, No. 7, 1986 (ppgs 583-589)

[GROS80] Gross, J.M., Jackson, P.E., Joyce, J and McGuire, F.A., "Distributed Database Design and Administration", *Distributed Databases*, Cambridge University Press, 1980 (ppgs 285-322)

[HEVN84] Hevner, Alan R., "Data Allocation and Retrieval In Distributed Systems", *Advances in Data Base Management*, Vol. 2, E.A. Unger, P.S. Fisher and J. Slonim (editors), Wiley Heyden Ltd, 1984 (ppgs 225 -252)

[IRAN79] Irani, Keki B; Khabbaz, Nicholas G, "A Model for a Combined Communication Network Design and File Allocation for Distributed Databases", *1st International Conference on Distributed Computing Systems*, Computer Society Press, 1979 (ppgs 15-21)

[LEVI79] Levin, Dan K. and Morgan, Howard Lee, "Optimizing Distributed Databases - A Framework for Research", *Distributed System Design*, IEEE Computer Society Press, Oct. 1979 (ppgs 321-326)

[MAHM76] Mahmoud, Samy and Riordon, J.S., "Optimal Allocation of Resources in Distributed Information Networks", *ACM Transactions on Database Systems*, Vol. 1, No.1, 1976 (ppgs 483-497)

[MARI84] Mariani, M.P., "Distributed Data Processing: Technology and Critical Issues", *TRW Series on Software Technology*, Vol. 4, 1984 (217 ppgs)

[MURO85] Muro, Shojiro; Ibaraki, Toshihide, Miyajima, Hiachiro and Hasegawa, Toshihara, "Evaluation of the File Redundancy in Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. SE-11, No.2, 1985 (ppgs 199 -204)

[NAVA84] Navathe, Shankant; Ceri, Stefano; Wiederhold, Gio and Dou, Jinglie, "Vertical Partitioning Algorithms for Database Design", *ACM Transactions on Database Systems*, Vol. 9, No. 4, 1984 (ppgs 680-710)

[PEEB81] Peebles, Richard and Manning, Eric, "System Architecture for Distributed Data Management", *IEEE Tutorial on Distributed Processing*, Third Edition, IEEE Computer Society Press, 1981 (pgs 451-464)

[PURK83] Purkayastha, S., Kar, G., Berelian, E., Wong, P., Casey, R.L., Farmer, L., Lo, P. and Chen, D., "Designing a Database Management System for Distributed Real Time Engineering Applications", *IEEE Transactions on Computers*, 1983 (ppgs 432-439)

[ROTH81] Rothnie, J.B. and Goodman, N., "A Survey of Research and Development in Distributed Database Management", *IEEE Tutorial on Distributed Processing*, Third Edition, IEEE Computer Society Press, 1981 (pgs 484-498).

[SCHN84] Schriederjans, Mark J., *Linear Goal Programming*, Petrocelli Books, Inc., 1984 (pgs 67-109).

[WOEL81] Woelk, Darrell W., "The Generation of Entity-Relationship Diagrams from User Documents", *A Master's Report*, Kansas State University, 1981.

[YU83] Yu, C.T., Siu, M.K., Iam, K. and Chen, C.H., "Adaptive File Allocation in Star Computer Network", *IEEE COMPSAC*, Nov 7-11, Computer Society Press, 1983 (ppgs. 537-545)

APPENDIX I
EXAMPLE DESIGN: DOCUMENT_HANDLER OUTPUT

Each DOCUMENT is labeled with a type (INPUT, OUTPUT, RESIDENT),frequency of use (i.e., per day), and has a document key

Each DATA_ELEMENT has an owner(s) associated with it (indicated by '*').

NEW_CUST

Document Attributes:
Location.output
Frequency: 5

*cust_no (doc.key)
*c_name
*c_str
*c_cty
*c_sta
*c_zip
*c_ph

CUST_FILE

Document Attributes:
Location.resident
Frequency: 5

c_no (doc.key)
c_name
c_str
c_cty
c_sta
c_zip
c_ph

CUST_LIST

Document Attributes:
Location.output
Frequency: 1

c_no (doc.key)
c_name
c_str
c_cty
c_sta
c_zip
c_ph

NEW_PART

Document Attributes:
Location.input
Frequency: 2

*part_no (doc.key)
*part_desc
*part_price

PART_SUP

Document Attributes:
Location.input
Frequency: 10

part_no (doc.key)
part_desc
*qty_made
*made_date
part_price

CALC_INV

Document Attributes:
Location.resident
Frequency: 30

part_no (doc.key)
qty_made
made_date
*made_to_date
qty_ord
cord_date
*ord_to_date
*qty_rem

INV_REPT

Document Attributes:
Location.output
Frequency: 1

part_no (doc.key)
part_desc
made_to_date
ord_to_date
qty_rem

CUST_ORD

Document Attributes:
Location.input
Frequency: 20

c_no (doc.key)
c_name
c_str
c_cty
c_sta
c_zip
c_ph
part_no
*qty_ord
*cord_date

HIST_FILE

Document Attributes:
Location.resident
Frequency: 30

c_no (doc.key)
part_no (doc.key)
cord_date (doc.key)
qty_ord
part_price
*tot_price

INVOICE

Document Attributes:
Location.output
Frequency: 10

c_no (doc.key)
c_name
c_str
c_cty
c_sta
c_zip
c_ph
part_no
part_price
qty_ord
tot_price
cord_date

APPENDIX II

EXAMPLE DESIGN: FUNCTIONAL DEPENDENCIES/BERN2 OUTPUT

**** FUNCTIONAL DEPENDENCIES CREATED FROM DOCUMENT_HANDLER ****

cust_no --> c_name, c_str, c_cty, c_sta, c_zip, c_ph

part_no --> part_desc

part_no --> part_desc, qty_made, made_date, part_price

part_no --> qty_made, made_date, made_to_date, qty_ord, ord_to_date, qty_rem

part_no --> part_desc, made_to_date, ord_to_date, qty_rem

c_no --> c_name, c_str, c_cty, c_sta, c_zip, c_ph, part_no, qty_ord, cord_date

c_no, part_no, cord_date --> qty_ord, part_price, tot_price

c_no --> c_name, c_str, c_cty, c_sta, c_zip, c_ph, part_no, part_price, qty_ord tot_price, cord_date

**** 3NF RELATIONS PRODUCED BY BERN2 ****

REL1 : c_no --> c_name, c_str, c_cty, c_sta, c_zip, c_ph, part_no, tot_price

REL2 : part_no --> qty_made, made_date, qty_ord, cord_date, part_desc, made_to_date, ord_to_date, qty_rem, part_price

APPENDIX III
EXAMPLE DESIGN: DIST_DATA

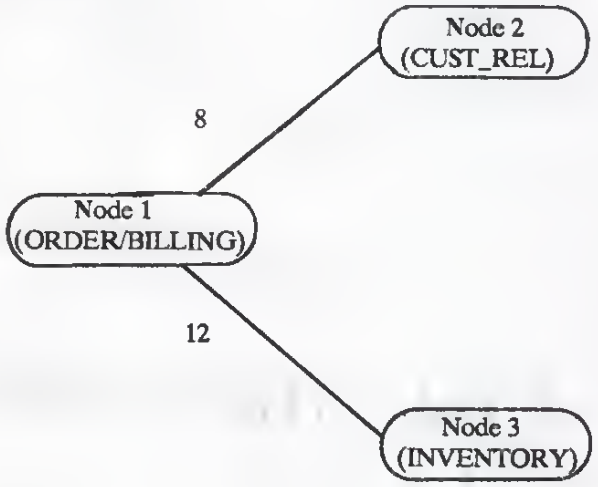
1) Identify Application Structure and Network Topology
(Algorithm Step 1)

PART A:

Enter No. Nodes on Network: 3
Enter No. of Organizations: 3

Enter Org. Name:	ORDER/BILLING	INVENTORY	CUST_REL
ORDER/BILLING on Node?	1	3	2
Enter Interconnect Nodes & Cost:	2,8	1,12	1,8
	3,12	—,—	—,—
	—,—	—,—	—,—
Enter Documents Owned:	CUST_ORD	NEW_PART	NEW_CUST
	HIST_FILE	PART_SUP	CUST_FILE
	INVOICE	CALC_INV	CUST_LIST
		INV_REPT	

Part B: Logical View of Network:



Part C: Build Cost Tables

(Nodes)

	1	2	3
1	0	8	12
2	8	0	20
3	12	20	0

2) Create Partitions/Files: Relation 1
(Algorithm Step 2)
Part A: Build Document Table (Sort by Node)

	c	c	c	c	c	c	c	part	tot
Node: Doc	no	name	str	cty	st	zip	ph	no	price
1 CUST_ORD	20	20	20	20	20	20	20	20	
1 HIST_FILE	30							30	30*
1 INVOICE	10	10	10	10	10	10	10	10	10
2 NEW_CUST	5*	5*	5*	5*	5*	5*	5*		
2 CUST_FILE	5	5	5	5	5	5	5		
2 CUST_LIST	1	1	1	1	1	1	1		
3 NEW_PART								2*	
3 PART_SUP								10	
3 CALC_INV								30	
3 INV_REPT								1	

Part B: Analyze Frequency of Use, Determine "Owners"

c_name
c_str
c_cty
c_st
c_zip
c_ph: 30 queries, 0 updates - Node 1 (X)
6 queries, 5 updates - Node 2
part_no: 60 queries, 0 updates - Node 1 (X)
41 queries, 2 updates - Node 3
tot_price: 10 queries, 20 udpates - Node 1 (X)

Part C: Merge Attributes by Owner to form Partitions:

File 1 = (c_no, c_name, c_str, c_cty, c_st, c_zip, _ph, part_no, tot_price)

(No partitioning done for this file)

Part D: Modify Partitions (if desired)

3) **Determine File Placement (Algorithm Step 3):** Relation 1, File Name = 1

PART A: Build Transaction Table

	Queries	Updates
1	30	30
2	6	5
3	41	2

PART B: Build Cost Tables

(Network Cost Table from Previous Step)

(Nodes)

	1	2	3
1	0	8	12
2	8	0	20
3	12	20	0

Queries

	1	2	3
1	0	240	360
2	48	0	120
3	492	820	0
	540	1060	480

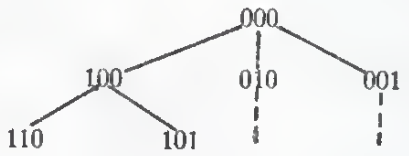
Updates

	1	2	3
1	0	240	360
2	40	0	100
3	24	40	0
	64	280	460

Total Costs: Node 1: 540 + 64 = 604
Node 2: 1060+280 = 1340
Node 3: 480 + 460 = 940

***** Recommended File Placement at Node 1 ****

4) Determine Profitability of Replication (Algorithm Step 4): File 1



a) Node 1 and 2:

Query Cost:	Update Cost:
Node 1,2 = 0	Node 1 = 240
Node 3 = 492	Node 2 = 40
	Node 3 = 64
Total = 492	Total = 344

Total Cost = 492 + 344 = 836 (No replication)

b) Node 1 and 3:

Query Cost:	Update Cost:
Node 1,3 = 0	Node 1 = 360
Node 2 = 48	Node 2 = 140
	Node 3 = 24
Total = 48	Total = 524

Total Cost = 48 + 524 = 572 ** Replicate **

c) Continue to Branch, Nodes 1,2,3:

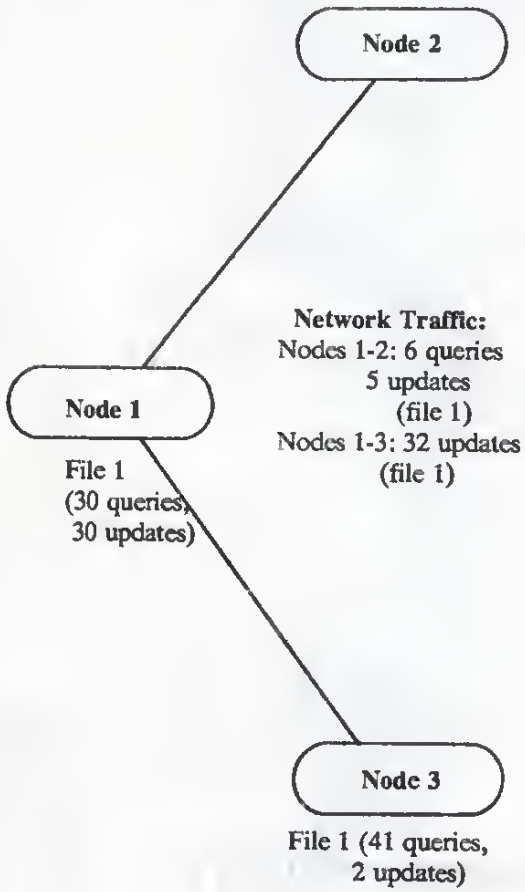
Query Cost:	Update Cost:
Nodes 1,2,3 = 0	Node 1 = 600
	Node 2 = 140
	Node 3 = 64
Total = 0	Total = 804

Total Cost = 0 + 804 = 804 (No Replication)

*** Recommended File Placement for File 1: NODE 1,NODE 3 ***
TOTAL COST = 572

5) Perform Sensitivity Analysis (if desired)
(Algorithm Step 5)

6) ** Final Placement for File 1 **



7) Create Partitions/Files: Relation 2
(Algorithm Step 2)

Part A: Build Document Table (Sort by Node)

							made	ord		
	part	qty	made	qty	cord	part	to	to	qty	part
Node:Doc	no.	made	date	ord	date	desc	date	date	rem	price
1 CUST_ORD	20			20*	20*					
1 HIST_FILE	30			30	30					30
1 INVOICE	10			10	10					10
3 NEW_PART	2*					2*				2*
3 PART_SUP	10	10*	10*			10				10
3 CALC_INV	30	30	30	30	30		30*	30*	30*	
3 INV_REPT	1					1	1	1		

Part B: Analyze Frequency of Use, Determine "Owners"

qty_made
made_date: 30 queries, 10 updates - Node 3 (X)

qty_ord
cord_date: 40 queries, 20 updates - Node 1 (X)
30 queries, 0 updates - Node 3

part_desc: 11 queries, 2 updates - Node 3 (X)

made_to_date
ord_to_date
qty_rem: 1 query, 30 updates - Node 3 (X)

part_price: 40 queries, 0 updates - Node 1 (X)
10 queries, 2 udpates - Node 3

Part C: Merge Attributes by Owner to form Partitions:

File 2 = (part_no, qty_ord, cord_date, part_price)
File 3 = (part_no, qty_made, made_date, part_desc,
made_to_date, ord_to_date, qty_rem)

Part D: Modify Partitions (if desired)

8) Determine File Placement: Relation 2, File Name= 2
(Algorithm Step 3)

Part A: Build Transaction Table: File 2 (algorithm step 4)
File 2 = (part_no, qty_ord, cord_date, part_price)

	Queries	Updates
1	40	20
2	0	0
3	40	2

Part B: Build Cost Tables

(Network Cost Table from Previous Step)

(Nodes)

	1	2	3
1	0	8	12
2	8	0	20
3	12	20	0

Queries

	1	2	3
1	0	320	480
2	0	0	0
3	480	800	0
	480	1120	480

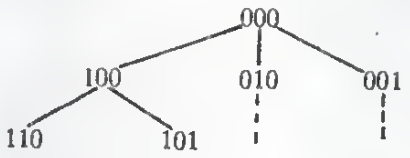
Updates

	1	2	3
1	0	160	240
2	0	0	0
3	24	40	0
	24	200	240

Total Costs: Node 1: 480 + 24 = 504
Node 2: 1120+200 = 1320
Node 3: 480 + 240 = 720

***** Recommended File Placement at Node 1 ****

9) **Determine Profitability of Replication:** File 2
(algorithm step 4)



a) Node 1 and 2:

Query Cost:	Update Cost:
Node 1,2 = 0	Node 1 = 160
Node 3 = 480	Node 2 = 200
	Node 3 = 64
Total = 480	Total = 224

Total Cost = 480 + 224 = 704 (No replication)

b) Node 1 and 3:

Query Cost:	Update Cost:
Node 1,3 = 0	Node 1 = 240
Node 2 = 0	Node 2 = 0
	Node 3 = 24
Total = 0	Total = 264

Total Cost = 0 + 264 = 264 ** Replicate **

c) Continue to Branch, Nodes 1,2,3:

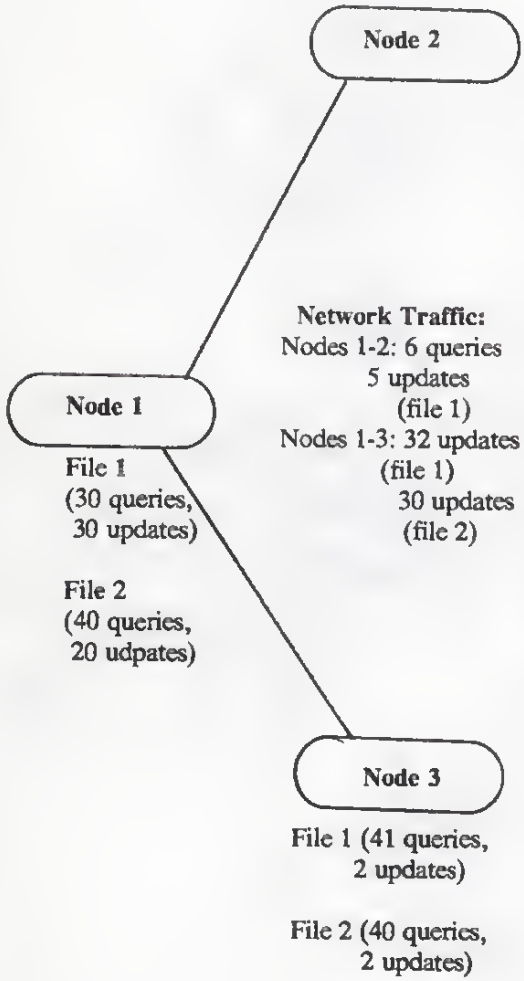
Query Cost:	Update Cost:
Nodes 1,2,3 = 0	Node 1 = 400
	Node 2 = 0
	Node 3 = 64
Total = 0	Total = 464

Total Cost = 0 + 464 = 464 (No Replication)

* Recommended File Placement for File 2: NODE 1,NODE 3 *
TOTAL COST = 264

10) Perform Sensitivity Analysis (if desired)
(Algorithm Step 5)

11) ** Final Placement For File 1 and File 2 **



12) **Determine File Placement:** Relation 2, File Name= 3
(Algorithm Step 3)

Part A: Build Transaction Table: File 3 (algorithm step 4)
File 3 = (part_no, qty_made, made_date, part_desc,
made_to_date, ord_to_date, qty_rem)

	Queries	Updates
1	0	0
2	0	0
3	1	42

Part B: Build Cost Tables

(Network Cost Table from Previous Step)

(Nodes)

	1	2	3
1	0	8	12
2	8	0	20
3	12	20	0

Queries

	1	2	3
1	0	0	0
2	0	0	0
3	12	20	0
	12	20	0

Updates

	1	2	3
1	0	0	0
2	0	0	0
3	504	840	0
	504	840	0

Total Costs: Node 1: 12 + 504 = 516
Node 2: 20 + 840 = 860
Node 3: 0 + 0 = 0

***** Recommended File Placement at Node 3 ****

13) Determine Profitability of Replication: File 3
(algorithm step 4)

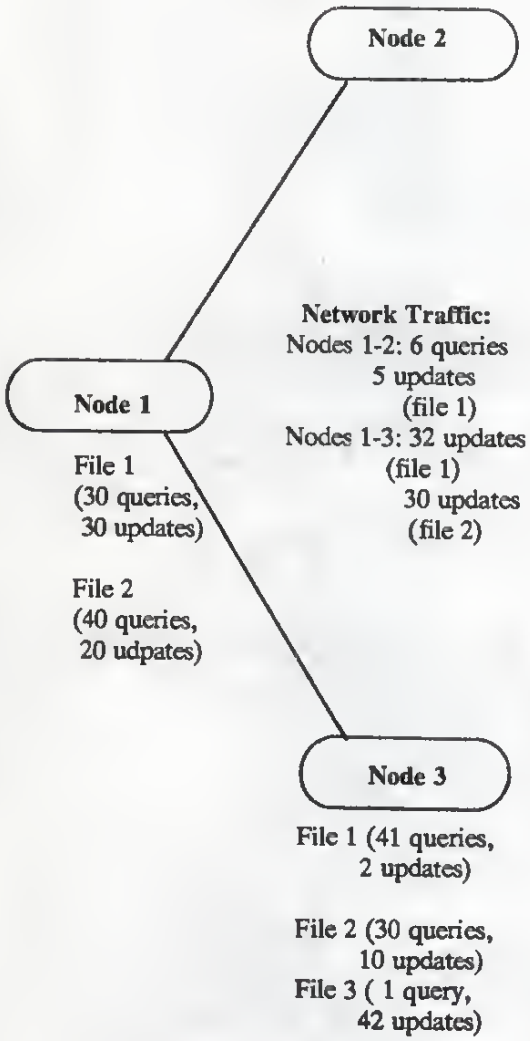
** Cost is zero: replication not profitable **

* Recommended File Placement for File 3: NODE 3 *
TOTAL COST = 0

14) Perform sensitivity analysis (if desired)
(Algorithm Step 5)

15) ** FINAL OUTPUT OF DIST_DATA TOOL **

File 1: at Nodes 1 and 3	Cost = 572
File 2: at Nodes 1 and 3	Cost = 264
File 3: at Nodes 2	Cost = 0
Total Cost	836



```
1
2
3   /*************************************************************************
4   *
5   *  DIST_DATA PROGRAM:
6   *
7   *     The dist_data program was designed to assist the database designer in determining
8   *     correct file partitioning and placement in a distributed database environment.
9   *     The program takes as input the following files:
10  *
11  *      - Bernfile:  This is a file produced by exectution of the 'bern2' program, which
12  *                   produces 3NF Relations
13  *      - Docfile:   This is a file produced by the 'dochandler' program that takes customer
14  *                   documents and converts them to Functional Dependencies
15  *
16  *     The program produces as output the following information:
17  *
18  *      - Recommended Partitioning of the 3NF Relations
19  *      - Recommended File Placement/Replication of these partitions onto nodes in a distributed
20  *        network
21  *
22  *     The main() function controls dist_data processing.  It solicits information from the
23  *     user regarding network nodes, organizational structure and document usage per organization.
24  *     Once this data is gathered it begins processing to determine file partitioning
25  *     and file placement.  The program consists of the following functions:
26  *
27  *      build_bern(): Builds the bern2 (3NF) relation table
28  *      build_rel(): Builds a relation table which indicates attribute usage on a per-node basis
29  *      find_least_cost(): Determines minimum cost connection between nodes on the network
30  *      build_parts(): Recommends file partitioning of the 3NF relations
31  *      pr_parts(): Prints resulting partitions created
32  *      file_placement(): Determines recommended file placment for each partition created
33  *                        (based on minimum cost for update/query usage)
34  *      branch_bound(): Performs branch and bound search to determine the profitability of
35  *                      replication
36  *      find_cost(): Finds least cost path between 2 nodes in a network
37  *      find_new():  Finds new cost associated with file replication
38  *      pr_reltbl(): Prints out contents of relation table
39  *      rebuild_cost(): Rebuilds the cost tables if sensitivity analysis has been requested
40  *                      (i.e., determine if replication is desirable if a 5%,10%,etc.
41  *                       communicationincrease/decrease is incurred)
42  *
43  *************************************************************************/
44
45  #include <stdio.h>
46  #include <string.h>
47  #include </c2/zebe/m.report/impl/dist_data.h>
48
```

```
49  int NO_NODES=0,NO_ATTR=0,NO_OOCS=0,NO_ORGS=0;
50  int NO_RELS=0,NO_ENTS=0,holdcmp=0;
51
52  main(argc,argv)
53  int argc;
54  char *argv[];
55  {
56
57      FILE *docptr, *bernptr, *fopen();
58      char *calloc();
59
60      int c=0,i=0,j=0,k=0,chkcmpr=0,tmpcost=0,tmpnode=0,relnum=0;
61      int search_node=0,least_cost=0,con_node = 0;
62      int relentry=0,no_parts=0,partnumb=0;
63      int tmp1_skip=0,tmp2_skip=0, operation = 0;
64      int flag=FALSE,flag2=FALSE, percent=0;
65      char sign;
66      char ans[4];
67      char tmporg[ORG_SZ];
68      char tmpdoc[OOC_SZ];
69
70      if(argc !=3)    /* Check for Valid No. of Args */
71      {
72          fprintf(stderr, "Usage: dist_data <docfile><bernfile>\n");
73          exit(1);
74      }
75      else if((docptr=fopen(*++argv,"r")) == NULL)
76      {
77          fprintf(stderr,"dist_data: cannot open %s\n",*argv);
78          exit(1);
79      }
80      else if((bernptr=fopen(*++argv,"r")) == NULL)
81      {
82          fprintf(stderr,"dist_data: cannot open %s\n",*argv);
83          exit(1);
84      }
85
86      /* GET TOTAL NO. OF OOCS/ATTRIBUTES FROM OOCFILE */
87
88      fscanf(docptr,"%d%d",&NO_OOCS,&NO_ATTR);
89
90
91      /* DETERMINE NUMBER OF BERN2 RELATIONS */
92
93      while ((c=fgetc(bernptr)) != EOF)
94      {
95          if(c=='(')
96              NO_RELS++;
```

```
97        }
98        fseek(bernptr,0L,0);
99
100       /*  INITIALIZE BERNTABLE  */
101
102       for(i=0; i<NO_RELS; i++)
103       {
104           for(j=0; j<NO_ATTR; j++)
105           {
106                strcpy(berntable[i].attributes[j].attrs,"");
107                berntable[i].attributes[j].key = FALSE;
108           }
109       }
110
111       /* BUILD BERN TABLE (holds bern2 relations)  */
112
113       build_bern(bernptr);
114
115       /*  QUERY USER  */
116
117       printf("Enter No. Nodes on Network:  ");
118       scanf("%d",&NO_NODES);
119       printf("\nEnter No. of Organizations:  ");
120       scanf("%d",&NO_ORGS);
121
122       /* INITIALIZE COST TABLE */
123
124       for(i=0; i<NO_NODES; i++)
125       {
126           for(j=0;j<NO_NODES;j++)
127               costable[i][j] = 0;
128       }
129
130       /* INITIALIZE PORTION OF RELATION TABLE  */
131
132       for(i=0; i< MAX_ENTRY; i++)
133       {
134           reltable[i].nodenum = 0;
135           strcpy(reltable[i].orgname,"");
136           strcpy(reltable[i].docname,"");
137       }
138
139       /* QUERY USER   */
140
141       for(i=0; i<NO_ORGS; i++)
142       {
143           strcpy(tmpdoc,"");
144           strcpy(tmporg,"");
```

```
145
146         printf("\nEnter Org. Name:    ");
147         scanf("%s",tmporg);
148
149         printf("\n%s on node?:    ",tmporg);
150         scanf("%d",&tmpnoda);
151
152         printf("\nEnter Interconnect Node(s) & Cost(s): Enter each node/cost on separate line.\n");
153         printf("                                        enter a '0' to complete\n");
154         for(j=0; j< NO_NODES; j++)
155         {
156             con_node=0;
157             scanf("%d",&con_node);
158
159             if(con_node != 0)
160             {
161                 scanf("%d",&tmpcost);
162                 costable[tmpnode-1][con_node-1] = tmpcost;
163                 costable[con_node-1][tmpnode-1] = tmpcost;
164                 tmpcost=0;
165             }
166             else
167             {
168                 break;
169             }
170         }    /* End of Nodes for Organization */
171
172         printf("\nEnter Documents Owned:  Enter each document on separte line\n");
173         printf("                          enter 'quit' to complete\n");
174         for(j=0; j<NO_DOCS; j++)
175         {
176             strcpy(tmpdoc,"");
177             scanf("%s",tmpdoc);
178             if((strcmp(tmpdoc,"quit") == 0) || (strcmp(tmpdoc,"QUIT") == 0))
179             {
180                 break;
181             }
182             else
183             {
184                 reltable[relentry].nodenum = tmpnode;
185                 strcpy(reltable[relentry].orgname,tmporg);
186                 strcpy(reltable[relentry].docname,tmpdoc);
187                 relentry++;
188             }
189         }   /* End of Organization Documents  */
190     }   /* End of Processing Organizations    */
191
192     NO_ENTS=relentry;
```

```
193
194       /* CHECK COST TABLE  */
195
196       for(i=0; i< NO_NODES; i++)
197       {
198           for(j=0; j<NO_NODES; j++)
199           {
200               if(costable[i][j] == 0)
201               {
202                   if (i != j)
203                   {
204                        search_node = j;
205                        tmpcost=find_least_cost(i,j,search_node,least_cost);
206                        costable[i][j] = tmpcost;
207                        costable[j][i] = tmpcost;
208                        least_cost = 0;
209
210                   }
211                   else
212                   {
213                        costable[i][j] = 0;
214                        costable[i][j] = 0;
215                   }
216               }
217           }
218       }   /* End of for Loop  */
219
220
221
222       /* DIST_DATA PROCESSING */
223
224       for(i=0; i< NO_RELS; i++)
225       {
226            /* INITIALIZE RELATION TABLE */
227
228           for (j=0; j<MAX_ENTRY; j++)
229           {
230               reltable[j].use_rel=FALSE;
231               reltable[j].total_query=0;
232               reltable[j].total_update=0;
233               for(k=0;k<MAX_ATTRS; k++)
234               {
235                   reltable[j].freq[k].key=FALSE;
236                   reltable[j].freq[k].owner=FALSE;
237                   reltable[j].freq[k].part_num=0;
238                   reltable[j].freq[k].frequency=0;
239               }
240           }
```

```
241         build_rel(docptr,i);
242         no_parts=build_parts(i);
243
244         for(k=0; k<MAX_PARTS; k++)
245         {
246           if(parts_created[k] != -1)
247           {
248             partnumb=parts_created[k];
249             file_placement(i,partnumb);
250             printf("\nPERFORMING REPLICATION ANALYSIS (Branch & Bound)...\n");
251             sleep(2);
252             branch_bound();
253
254             /* SEE IF SENSITIVITY ANALYSIS DESIRED */
255
256             while (flag == FALSE)
257             {
258                 printf("\nSensitivity Analysis Desired ?  ");
259                 scanf("%s",ans);
260                 if((ans[0] == 'y') || (ans[0] == 'Y'))
261                 {
262                     flag=TRUE;
263
264                     /* RE-INITIALIZE SOLUTION ARRAY */
265
266                     for (j=0; j< MAX_NODES; j++)
267                     {
268                          if(j != svnode - 1)
269                         {
270                             solution_array[j] = 0;
271                         }
272                     }
273                     printf("\nInput will be taken as a percentage increase/decrease\n");
274                     while (flag2 == FALSE)
275                     {
276                         printf("\nEnter '+' for Increase/ '-' for Decrease: ");
277                         scanf("%1s",&sign);
278                         if((sign == '+') || (sign == '-'))
279                             flag2=TRUE;
280                         else
281                             printf("Must indicate '+' or '-' (increase or decrease)\n");
282                     }
283                     printf("Enter Cost Factor (i.e., 5, 50, etc): ");
284                     scanf("%d",&percent);
285
286                     rebuild_cost(sign,percent);
287                     branch_bound();
288                 }
```

```
289                 else if((ans[0] == 'n') || (ans[0] == 'N'))
290                     flag=TRUE;
291                 else
292                     printf("Answer must be Yes or No\n");
293               }
294               flag2=FALSE;
295             }
296             flag=FALSE;
297           }
298
299           /* RESET FILE POINTER FIRST DOCUMENT (strip off total
300             document and attribute counters)                    */
301
302           fseek(docptr,0L,0);
303           fscanf(docptr,"%d%d",&tmp1_skip,&tmp2_skip);
304
305       }  /* End of Relations  */
306
307       /*  COMMENT OUT PRINT OF THESE FILES
308           pr_costbl();
309           pr_berntbl();
310           */
311   }  /* End of Main */
312
313
314   /*************************************************************************
315   *  BUILD_BERN():
316   *
317   *    Processing: Parses the bern2 file to determine 3NF relations, and to flag
318   *                which attributes are key fields in the relation
319   *
320   *    Input:      File pointer to output file produced by bern2 (3NF Relations)
321   *
322   *    Functions
323   *    Called:     (None)
324   *
325   *    Output/
326   *    Return:     Populates a structure which contains the bern2 relations for
327   *                later use in partitioning functions
328   *************************************************************************/
329
330   build_bern(fp)
331   char *fp;
332   {
333
334           int relnum=0,attrct=0,i=0;
335
336           int keyflag = TRUE;
```

```
337         int validchar = FALSE;
338
339         int holdchar;
340         holdchar = fgetc(fp);   /* Get rid of leading '('  */
341         while (holdchar != EOF)
342         {
343             holdchar=fgetc(fp);
344             switch(holdchar)
345             {
346                 case '\n':
347                 case '\t':
348                 case '>':
349                       validchar=FALSE;
350                       break;
351                 case '(':
352                       keyflag = TRUE;
353                       attrnum[relnum] = attrct ;
354                       relnum++;
355                       attrct=0,i=0;
356                       validchar=FALSE;
357                       break;
358                 case ')':
359                       keyflag=FALSE:
360                       validchar=FALSE;
361                       break;
362                 case ' ':
363                       if((validchar==TRUE) && (keyflag==TRUE))
364                       {
365                           berntable[relnum] attributes[attrct] key = TRUE;
366                       }
367                       if(validchar==TRUE)
368                       {
369                           berntable[relnum].attributes[attrct].attrs[i] = '\0';
370                           attrct++;
371                           i=0;
372                       }
373                       validchar=FALSE;
374                       break;
375                 case EOF:
376                       if((validchar==TRUE) && (keyflag==TRUE))
377                       {
378                           berntable[relnum].attributes[attrct].key = TRUE;
379                       }
380                       if(validchar==TRUE)
381                       {
382                           berntable[relnum].attributes[attrct].attrs[i] = '\0';
383                           attrct++;
384                           i=0;
```

```
                        }
                        attrnum[relnum] = attrct;
                        break;
                default:
                        berntable[relnum].attributes[attrct].attrs[i] = holdchar;
                        validchar=TRUE;
                        i++;
              }  /* End of "switch"  */
         }  /* End of "while"  */
}

/*****************************************************************************
*  FIND_LEAST_COST():
*
*    Processing: Recursive function that traces path connections between nodes
*                on the network, to determine the "least cost" path between
*                any given pair of nodes (the dist_data processing requires
*                a fully connected network (virtual) in order to have a cost
*                associated between any two nodes.  This is needed for proper
*                file placement based on minimum cost).
*
*    Input:      Index values (i,j) of network cost_table, search_node which
*                which is the node representing an "open" connection and
*                min_cost which is the placeholder for the minimum cost found
*                so far in the search
*
*    Functions
*    Called:     find_least_cost()  (Recursive function)
*
*    Output/
*    Return:     Returns the least cost connection between a given pair of nodes
*****************************************************************************/

find_least_cost(i,j,search_node,mincost)
int i,j,search_node,mincost;
{

        int k, new_cost,add_cost = 0;
        int NO_NODES = 5;

        for(k=0; k<NO_NODES;k++)
        {
           if((costable[search_node][k] !=0) && (k != j))
           {
                add_cost = add_cost + costable[search_node][k];
                if(costable[i][k] !=0)
                {
                        add_cost=add_cost + costable[i][k];
```

```
433                             if(mincost == 0)
434                                     mincost = add_cost;
435                             else if(add_cost < mincost)
436                                     mincost = add_cost;
437                             aod_cost=0;
438                     }
439                     else
440                     {
441                             search_node = k;
442                             find_least_cost(i,j,search_node,add_cost,mincost);
443                     }
444             }
445         }
446         return(mincost);
447 }
448
449 /*****************************************************************************
450 *  BUILD_REL():
451 *
452 *     Processing: Parses the bern2 file to determine 3NF relations, and to flag
453 *                 which attributes are key fields in the relation.
454 *
455 *     Input:      File pointer to output file produced by bern2 (3NF Relations)
456 *
457 *     Functiona
458 *     Called:     (None)
459 *
460 *     Output/
461 *     Return:     Populates a structure which contains the barn2 relations for
462 *                 later use in partitioning functions
463 *****************************************************************************/
464
465 build_rel(fp,relno)
466 char *fp;
467 int relno;
468 {
469
470
471     int validchar=FALSE,attrowner=FALSE,keyflag=FALSE;
472     char tmpdoc[DOC_SZ];
473     char tmpattr[ATR_SZ];
474     char tmpjunk[DOC_SZ];
475     int check=0, dumpchar=0;
476     int tmpfreq=0,attrct=0,tblent=0,attrsz=0,i=0;
477
478     strcpy(tmpattr,"");
479     dumpchar = fgetc(fp);
480
```

```
481     while (check != EOF)
482     {
483         check=fgetc(fp);
484         if(check == '+')
485         {
486             fscanf(fp,"%s",tmpdoc);
487
488          /* SKIP THE "MISC" IN DOCUMENT LIKE LOCATION.INPUT, ETC  */
489
490             for (i=0; i< 4; i++)
491             {
492                 fscanf(fp,"%s",tmpjunk);
493             }
494             fscanf(fp,"%d",&tmpfreq);
495             check=fgetc(fp);
496         }
497         else
498         {
499             switch(check)
500             {
501                 case '\t':
502                 case '\n':
503                 case ' ':
504                 case '*':
507                     {
508                         tmpattr[attrsz] = '\0';
509
510                         for(attrct=0; attrct< (attrnum[relno] +1); attrct++)
511                         {
512                             if(strcmp(tmpattr,berntable[relno].attributes[attrct].attrs) == 0)
513                             {
514                                 if(berntable[relno].attributes[attrct].key == TRUE)
515                                     keyflag=TRUE;
516                                     for(tblent=0; tblent < NO_ENTS+1; tblent++)
517                                     {
518                                         if(strcmp(tmpdoc,reltable[tblent].docname) == 0)
519                                         {
520                                             reltable[tblent].use_rel=TRUE;
521                                             reltable[tblent].freq[attrct].frequency=tmpfreq;
522                                             if(attrowner == TRUE)
523                                             {
524                                                 reltable[tblent].freq[attrct].owner = TRUE;
525                                             }
526                                             if(keyflag==TRUE)
527                                                 reltable[tblent].freq[attrct].key=TRUE;
528                                         }
529                                     } /* Finish search of table for one attr */
530                             } /* Finish "if" statement */
```

```
531                                     }  /* Finish "for" statement */
532                                     validchar=FALSE;
533                                     attrsz=0;
534                                     strcpy(tmpattr,"");
535                                     attrowner = FALSE;
536                                     keyflag=FALSE;
537                                 }
538                             break;
539
540                         case '%':
541                             attrowner=TRUE;
542                             break;
543                         default:
544                             tmpattr[attrsz] = check;
545                             attrsz++;
546                             validchar=TRUE;
547                             break;
548
549                     }   /* End of switch statement */
550                 }  /* End of "else" statement  */
551             }  /* End of "while" statement  */
552     }
553
554     /*****************************************************************************
555     *  BUILD_PARTS():
556     *
557     *    Processing: Analyzes each attribute in the relation table to determine
558     *                how often this attribute is used at each node.  It then
559     *                "assigns" the attribute to the node where it is most frequently
560     *                used.  After all attributes are processed, the attributes are
561     *                grouped together by frequency of use (i.e., all attributes which
562     *                are used most frequently at node 'x' will be grouped together, thus
563     *                forming partitions.  After the partitioning has been completed,
564     *                this function allows the user to make modifications to the partitioning
565     *                if desired (The results are displayed and the user may CREATE a
566     *                new partition, move an attribute from one partition to another or
567     *                MERGE two partitions)
568     *
569     *    Input:      Relation number which is to be processed, or analyzed for partitioning
570     *
571     *    Functions
572     *    Called:     pr_parts()
573     *
574     *    Output/
575     *    Return:     Populates the relation table with a partition number for each attribute
576     *                and returns the number of partitions created.
577     *****************************************************************************/
578
```

```
build_parts(relno)
int relno;
{
    struct ATTCT
    {
        int t_qry;
        int t_upd;
    }attct[MAX_NODES];


    int count=0,tblent=0,NODE_NUM=0,keypno=0,chk_freq=0,i=0,most_freq=0;
    int node=0,m_upd=0,numb_parts=0;
    int flag=FALSE,modflag=FALSE,found_newnumb=FALSE,NEW_PART=TRUE;
    int chkflag1=FALSE,chkflag2=FALSE,ansflag=0,att_number=0,move_part=0;
    int npart_no=0,part1=0,part2=0, operation = 0;
    char ans[4], attribute[ATR_SZ];



    keypno=NO_NODES+1;

    /* INITIALIZE PARTITION ARRAY */

    for (i=0; i< MAX_PARTS; i++)
        parts_created[i] = -1;

    /* DISPLAY RESULTS TO USER AS ANALYSIS IS BEING PERFORMED */

    printf("\n\nANALYZING ATTRIBUTE USAGE FOR RELATION %d ....\n", (relno+1));

    /* LOOP THROUGH ALL ATTRIBUTES TO ANALYZE  */

    for(count=0; count< (attrnum[relno]+1); count++)
    {

        /* INITIALIZE ATTCT     */

        for(i=0;i<MAX_NODES;i++)
        {
            attct[i].t_qry=0;
            attct[i].t_upd=0;
        }

        /* IF ATTRIBUTE IS A KEY -- IT IS NOT INCLUDED IN ANALYSIS */
        /* (as a key will appear in all partitions created)          */

        if(reltable[0].freq[count].key != TRUE)
        {
```

```
627         /* LOOP THROUGH TABLE ENTRIES FOR FREQ. OF USE */
628
629         for(tblent=0; tblent<NO_ENTS+1; tblent++)
630         {
631             NODE_NUM=reltable[tblent].nodenum -1;
632             if(reltable[tblent].freq[count].owner == TRUE)
633                 attct[NODE_NUM].t_upd = attct[NODE_NUM].t_upd + reltable[tblent].freq[count].frequency;
634             else
635                 attct[NODE_NUM].t_qry = attct[NODE_NUM].t_qry + reltable[tblent].freq[count].frequency;
636         }
637     }
638
639     /*  DETERMINE "OWNER" (Node most frequently accessed) */
640
641     most_freq=0,m_upd=0,node=0;
642
643     /* PRINT FOR USER */
644
645     if(reltable[0].freq[count].key != TRUE)
646         printf("\n%s:    ",berntable[relno].attributes[count].attrs);
647         for(i=0; i<NO_NODES;i++)
648         {
649                 if(reltable[0].freq[count].key != TRUE)
650                 printf("\n\t%d  queries\t%d updates (Node %d)",attct[i].t_qry,attct[i].t_upd,i+1);
651             chk_freq = attct[i].t_upd + attct[i].t_qry;
652             if(chk_freq > most_freq)
653             {
654                 most_freq = chk_freq;
655                 m_upd = attct[i].t_upd;
656                 node = i;
657             }
658             else if (chk_freq == most_freq)
659             {
660                 if (attct[i].t_upd > m_upd)
661                 {
662                     most_freq = chk_freq;
663                     m_upd = attct[i].t_upd;
664                     node = i;
665                 }
666             }
667         }
668
669     /* POPULATE ATTRIBUTE WITH PARTITION NUMBER     */
670     /* (Partition number equals owner node no.      */
671
672     for(tblent=0;tblent<NO_ENTS+1;tblent++)
673     {
674         if(reltable[0].freq[count].key != TRUE)
```

```
675                 {
676                     reltable[tblent].freq[count].part_num=node+1;
677                 }
678                 else
679                 {
680                     reltable[tblent].freq[count].part_num = keypno;
681                 }
682             }
683
684             /* KEEP TRACK OF NUMBER OF PARTITIONS CREATED */
685
686             NEW_PART=TRUE;
687             for(i=0; i< NO_NODES; i++)
688             {
689                 if(node+1 == parts_created[i])
690                     NEW_PART=FALSE;
691             }
692
693             if(NEW_PART == TRUE)
694             {
695                 parts_created[numb_parts] = node+1;
696                 numb_parts++;
697             }
698         }
699
700         /*  PRINTOUT RESULTING PARTITIONS CREATED */
701         pr_parts(numb_parts,relno);
702
703         /* SEE IF DISPLAY OF PARTITIONING IS DESIRED */
704             while (flag == FALSE)
705             {
706
707                 printf("\nDisplay of Partitioning Analysis Desired ?  ");
708                 scanf("%s",ans);
709                 if((ans[0] == 'y') || (ans[0] == 'Y'))
710                 {
711                     flag=TRUE;
712                     pr_reltbl(relno);
713                 }
714                 else if((ans[0] == 'n') || (ans[0] == 'N'))
715                     flag=TRUE;
716                 else
717                     printf("Answer must be Yes or No\n");
718             }
719
720             /* SEE IF MODIFICATIONS TO PARTITIONING ARE DESIRED */
721
722             flag = FALSE;
```

```
723         while (flag == FALSE)
724         {
725             printf("\nModification of Partitioning Desired ?  ");
726                 scanf("%s",ans);
727                 if((ans[0] == 'y') || (ans[0] == 'Y'))
728                 {
729                     flag=TRUE;
730                     modflag=TRUE;
731                 }
732                 else if((ans[0] == 'n') || (ans[0] == 'N'))
733                     flag=TRUE;
734                 else
735                     printf("Answer must be Yes or No\n");
736         }
737
738         while(modflag==TRUE)
739         {
740             operation = 0;
741             printf("Operations Available:\n");
742             printf("1 = MERGE two partitions\n");
743             printf("2 = MOVE an attribute to a different partition\n");
744             printf("3 = CREATE a new partition\n");
745             printf ("4 = quit (done making modifications\n");
746             printf("\nEnter Number of Desired Operation:  ");
747             scanf("%d",&operation);
748             switch(operation)
749             {
750                 case 1:
751                       ansflag = FALSE, chkflag1 = FALSE, chkflag2 = FALSE;
752                       part1= 0,part2 = 0;
753                       if(numb_parts == 1)
754                       {
755                         printf("\nInvalid operation -- only 1 partition exists\n\n");
756                         operation = 0;
757                       }
758                       else
759                       {
760                       while (ansflag == FALSE)
761                       {
762                           printf("This Operation will merge two existing partitions into one\n");
763                           printf("in the form of MERGE PARTITION ___ WITH ___\n\n");
764                           printf("Enter File Partition Numbers: ");
765                           scanf("%d%d",&part1,&part2);
766                           for(i=0; i<numb_parts; i++)
767                           {
768                               if(part1 == parts_created[i])
769                                   chkflag1=TRUE;
770                               if(part2 == parts_created[i])
```

```
771                             chkflag2=TRUE;
772                         }
773
774                         if((chkflag1 == TRUE) && (chkflag2 == TRUE))
775                             ansflag = TRUE;
776                         else
777                         {
778                             ansflag = FALSE;
779                             if(chkflag1 == FALSE)
780                                 printf("First Partition Entered is invalid partition number\n");
781                             else
782                                 printf("Second Partition Entered is invalid partition number\n");
783                         }
784                     )
785                     }
786                     break;
787
788                 case 2:
789
790                     ansflag = FALSE, chkflag1 = FALSE, chkflag2 = FALSE, move_part=0;
791                     while (ansflag == FALSE)
792                     {
793                         printf("This Dparation will move an attribute from one partition to another\n")
794                         printf(": in the form of MOVE ATTRIBUTE ___ TO PARTITION ___\n");
795                         printf("Enter Attribute Name:  ");
796                         scanf("%s",attribute);
797                         printf("Enter Partition Number:  ");
798                         scanf("%d",&move_part);
799
800                         /* VERIFY A VALID ATTRIBUTE WAS ENTERED */
801
802                         for(i=0; i< (attrnum[relno] +1); i++)
803                         (
804                             if(strcmp(attribute,barntable[relno].attributes[i].attrs) == 0)
805                             {
806                                 chkflag1 = TRUE;
807                                 att_number= i;
808                             }
809                         )
810
811                         /* VERIFY VALID PARTITION NUMBER WAS ENTERED */
812
813                         for(i=0; i<numb_parts; i++)
814                         (
815                             if(move_part == parts_created[i])
816                             {
817                                 chkflag2=TRUE;
818                             )
```

```
819                         }
820
821                         if((chkfleg1 == TRUE) && (chkflag2 == TRUE))
822                            ansfleg = TRUE;
823                         else
824                         {
825                             ansflag = FALSE;
826                             if(chkflag1 == FALSE)
827                                 printf("Invalid attribute name entered\n");
828                             else if (chkflag2 == FALSE)
829                                 printf("Partition Entered is invalid partition number\n");
830                         }
831                     }
832                     break;
833
834               case 3:
835
836                     printf("This Operation creates a new file partition\n");
837                     printf("This partition can then be populated by a series of MOVE operations\n\n");
838                     while(found_newnumb==FALSE)
839                     {
840                        npart_no++;
841                        found_numb=TRUE;
842                        for(i=0; i<numb_parts;i++)
843                        {
844                           if((npart_no == NO_NODES +1) || (npart_no == parts_created[i]))
845                               found_newnumb=FALSE;
846                        }
847                     }
848                     parts_created[numb_parts] = npart_no;
849                     printf("*** PARTITION NUMBER CREATED = %d ***\n",npart_no);
850                     sleep(2);
851                     ansflag=TRUE;
852                     numb_parts++;
853                     break;
854
855               cese 4:
856                     modflag = FALSE;
857                     ansflag = FALSE;
858                     break;
859
860               default:
861                     printf("Operation must be Numbers 1-4 \n");
862
863
864           }  /* End of "switch" statement */
865
866       /* CHECK OPERATION REQUESTED AND MAKE APPROPRIATE CHANGES TO RELTABLE */
```

```
867
868             if((operation == 1) || (operation == 2))
869             {
870                 for (tblent=0; tblent < NO_ENTS +1; tblent ++)
871                 {
872                     if(operation == 2)
873                         reltable[tblent].freq[att_number].part_num = move_part;
874
875                     if(operation == 1)
876                     {
877                         for(count=0; count < (attrnum[relno] +1); count ++)
878                         {
879                             if(reltable[tblent].freq[count].part_num == part1)
880                             {
881                                 reltable[tblent].freq[count].part_num = part2;
882                             }
883                         }
884                     }
885                  }
886             }
887             if(operation == 1)
888             {
889                 for(i=0; i< numb_parts; i++)
890                 {
891                     if(parts_created[i] == part1)
892                     {
893                         parts_created[i] = -1;
894                         numb_parts--;
895                     }
896                 }
897             }
898
899             if(ansflag == TRUE)
900             {
901                 printf("\nFILE PARTITIONS AFTER MODIFICATIONS\n");
902                 pr_parts(numb_parts,relno);
903             }
904         }  /* End of "while" statement */
905         return(numb_parts);
906     }
907
908
909     /*****************************************************************************
910     *  PR_PARTS():
911     *
912     *    Processing: Prints out the file partitions created for a given relation
913     *
914     *    Input:      Relation number and number of partitions created
```

```
915   *
916   *     Functions
917   *     Called:      (None)
918   *
919   *     Output/
920   *     Return:      simply prints to output
921   ********************************************************************************/
922
923   pr_parts(no_parts,relnumb)
924   int no_parts,relnumb;
925   {
926
927       int attrno, i;
928       unsigned int part_pr;
929       int PART_FOUNO = FALSE;
930
931       printf("\n*********************************************************************************\n\n");
932       printf("FILE PARTITION(S) CREATED FOR RELNO %d\n",relnumb+1);
933       printf("(File Partition Names Assigned According To 'Node/Owner')\n");
934
935       for (i=0; i< MAX_PARTS; i++)
936       {
937
938          part_pr = parts_created[i];
939          if(part_pr != -1)
940          {
941            printf("\nFILE PART %d: ",part_pr);
942            for(attrno=0; attrno < (attrnum[relnumb] +1); attrno++)
943            {
944                if(reltable[0].freq[attrno].part_num==NO_NODES + 1)
945                {
946                    printf("%s > ",berntable[relnumb].attributes[attrno].attrs);
947                }
948                else if(part_pr == reltable[0].freq[attrno].part_num)
949                    printf(" %s",berntable[relnumb].attributes[attrno] attrs);
950
951             }
952             printf("\n\n*********************************************************************************\n\n");
953       }
954       }
955   }
956
957   /*****************************************************************************
958   *  FILE_PLACEMENT():
959   *
960   *     Processing: Works on one file partition at a time, to build transaction table
961   *                 (eggregate update/query access to this partition from each node),
962   *                 builds the associated transaction cost tables (query and update
```

```
963   *                costs to this file if placed on any given node), and determines
964   *                recommended file placement based on least-cost.
965   *
966   *    Input:      The relation number and partition number to be analyzed for file placement
967   *
968   *    Functions
969   *    Called:     (None)
970   *
971   *    Output/
972   *    Return:     Sets file placement bit for this partition in global solution_array
973   *                (returns nothing)
974   ********************************************************************************/
975
976   file_placement(relnum,part_no)
977   int relnum,part_no;
978   {
979
980
981       int i=0,j=0,tblent=0,count=0,svfreq=0,UPD_TRANS=FALSE; int calc_total=0,calc_upd=0,calc_qry=0,
982       svnode=0, svmin_cost=0;
983
984       /* INITIALIZE TRANSACTION/COST TABLES/ SOLUTION ARRAY */
985
986       for (i=0; i<MAX_NODES; i++)
987       {
988           trans_tbl[i].t_qry=0;
989           trans_tbl[i].t_upd=0;
990           solution_array[i] =0;
991
992           for(j=0; j<MAX_NODES; j++)
993           {
994               qry_cost[i][j] = 0;
995               upd_cost[i][j] = 0;
996           }
997       }
998
999       /* SEARCH RELTABLE TO FIND TOTAL UPD AND QUERY FOR EACH NODE */
1000      /*  (Keys are included in total access )                      */
1001
1002      for(tblent=0; tblent < NO_ENTS +1; tblent++)
1003      {
1004          svnode = reltable[tblent].nodenum - 1;
1005          for(count=0; count < attrnum[relnum] +1; count++)
1006          {
1007              if(reltable[tblent].freq[count].part_num==part_no)
1008              {
1009                  if(reltable[tblent].freq[count].owner == TRUE)
1010                      UPD_TRANS=TRUE;
```

```
1011                    if(svfreq < reltable[tblent].freq[count].frequency)
1012                        svfreq = reltable[tblent].freq[count] frequency;
1013                }
1014            }
1015
1016            if(UPD_TRANS == TRUE)
1017            {
1018                trans_tbl[svnode].t_upd = trans_tbl[svnode].t_upd + svfreq;
1019                    UPD_TRANS=FALSE;
1020            }
1021            else
1022                trans_tbl[svnode].t_qry = trans_tbl[svnode].t_qry + svfreq;
1023                    svfreq= 0;
1024        }
1025
1026        /* PRINT TRANSACTION TABLE  */
1027
1028        printf("\n\nTRANSACTION TABLE FOR RELND:%d  FILE PARTITION: %d\n\n",relnum+1,part_no);
1029        printf("\tNODE\tQUERY\tUPDATE\n");
1030
1031        for(i=0; i<ND_NODES; i++)
1032            printf("\t  %d\t %d\t  %d\n",i+1,trans_tbl[i].t_qry,trans_tbl[i].t_upd);
1033
1034            /* BUILD COST TABLES FOR QUERIES AND UPDATES  */
1035
1036            for(i=0;i<ND_NODES;i++)
1037            {
1038                for(j=0;j<ND_NODES;j++)
1039                {
1040                    qry_cost[i][j] = trans_tbl[i].t_qry * costable[i][j];
1041                    upd_cost[i][j] = trans_tbl[i].t_upd * costable[i][j];
1042                }
1043            }
1044
1045
1046            /* DETERMINE RECOMMENDED FILE PLACEMENT FOR FILE PARTITION */
1047
1048            printf("\nTOTAL COSTS ASSOCIATED WITH FILE PLACEMENT:\n\n");
1049
1050            for(j=0;j<ND_NODES;j++)
1051            {
1052                for(i=0;i<ND_NODES;i++)
1053                {
1054                    calc_qry = qry_cost[i][j] + calc_qry;
1055                    calc_upd = upd_cost[i][j] + calc_upd;
1056                }
1057                calc_total = calc_qry + calc_upd;
1058                printf("NODE %d:  %d Query + %d Update = %d Total Cost\n",j+1,calc_qry,calc_upd,calc_total);
```

```
1059
1060                /* CHECK TO SEE IF MINIMUM COST  */
1061
1062                if(j==0)
1063                {
1064                    svnode = j+1;
1065                    min_cost= calc_total;
1066                }
1067                else if(calc_total < min_cost)
1068                {
1069                    min_cost=calc_total;
1070                    svnode = j+1;
1071                }
1072                calc_qry=0,calc_upd=0,calc_total=0;
1073            }
1074
1075        svmin_cost = min_cost;
1076
1077        printf("\n**** RECOMMENDED FILE PLACEMENT AT NODE %d, TOTAL COST= %d ***\n\n",svnode,min_cost);
1078        sleep(5);
1079
1080        /* SET FILE PLACEMENT BIT FOR THIS NODE  */
1081
1082        solution_array[svnode -1] = 1;
1083    }
1084
1085    /*****************************************************************************
1086    *  BRANCH_BOUND():
1087    *
1088    *    Processing: Analyzes profitability of replication, by performing a branch and
1089    *                bound search.  After a single file placement has been determined,
1090    *                this function will see if 2 copies of the file are profitable, 3 copies,
1091    *                etc   If at any point, the cost of replication is larger than the
1092    *                minimum cost, the "branching" stops for this node.  Should a total
1093    *                cost of placing copies of the file at multiple nodes be less than
1094    *                the previous minimum; replication is deemed profitable   The results
1095    *                of the replication analysis (and associated) costs are displayed to
1096    *                the user.
1097    *
1098    *    Input:      None
1099    *
1100    *    Functions
1101    *    Called:     find_cost()
1102    *
1103    *    Output/
1104    *    Return:     Updates global solution array to set new file placement bit(s) if
1105    *                replication was found to be profitable.
1106    *****************************************************************************/
```

```
1107
1108  branch_bound()
1109  {
1110      int keep_cost[MAX_NODES];     /* Keep intermediate cost figures */
1111
1112      int i=0, node=0, level=1, new_cost=0, mincost_node=0, CONTINUE_BRANCH=TRUE;
1113
1114      /* INITIALIZE KEEP_COST ARRAY */
1115
1116      for(node=0; node < MAX_NODES; node++)
1117          keep_cost[node] = 0;
1118
1119      /* MAIN ENTRY FOR BRANCH AND BOUND SEARCH  */
1120
1121      while((CONTINUE_BRANCH == TRUE) && (level < NO_NODES))
1122      {
1123          /* RESET BRANCH FLAG FOR NEXT PASS  */
1124          /* MOVE DOWN 1 LEVEL                */
1125
1126              CONTINUE_BRANCH= FALSE;
1127              level++;
1128
1129              for(node=0; node < NO_NODES; node ++)
1130              {
1131                  if(solution_array[node] != 1)
1132                  {
1133                    new_cost=find_cost(node);
1134                    if(((level==2)&&(new_cost<svmin_cost))||((level!=2)&&(new_cost<min_cost)))
1135                    {
1136                       min_cost = new_cost;
1137                       mincost_node= node;
1138                       CONTINUE_BRANCH=TRUE;
1139                    }
1140                  }
1141              }
1142
1143              if(CONTINUE_BRANCH == TRUE)
1144                  solution_array[mincost_node] = 1;
1145
1146          }
1147          printf("\n**** AFTER REPLICATION ANALYSIS, RECOMMENDED FILE PLACEMENT: ****\n");
1148
1149          for(i=0; i< NO_NODES; i++)
1150          {
1151              if(solution_array[i] == 1)
1152                  printf("\tNODE %d\n",i+1);
1153          }
1154          printf("\n**** TOTAL COST: %d \n",min_cost);
```

```
1155  }
1156
1157  /*****************************************************************************
1158  *  FIND_COST():
1159  *
1160  *    Processing: Called by the branch_bound function, this function computes the
1161  *                new costs associated with file replication.  This is done by parsing
1162  *                the quary and update cost tables - keeping the proposed replicaion
1163  *                of a partition in mind (i.e., if the file were placed on 2 nodes
1164  *                queries are now satisfied (or free) at both these nodes, the other
1165  *                nodes will access the file copy from the node representing tha
1166  *                least cost.  Updates on the other hand would be higher as each copy
1167  *                of the fila must be updatad).
1168  *
1169  *    Input:      The node being analyzed for replication
1170  *
1171  *    Functions
1172  *    Called:     (None)
1173  *
1174  *    Output/
1175  *    Return:     Returns the cost associated with the proposed replication
1176  *****************************************************************************/
1177
1178  find_cost(new_node)
1179  int new_node;
1180  {
1181
1182
1183      int calc_qry = 0, calc_upd = 0, tot_cost = 0;
1184      int i, j;
1185
1186
1187      /* DETERMINE QUERY COST WITH NEW FILE PLACEMENT  */
1188
1189      for(i=0; i< NO_NODES; i++)
1190      {
1191          if(( i != new_node) && (solution_array[i] != 1))
1192          {
1193              calc_qry = qry_cost[i][new_node];
1194              for(j=0; j< NO_NODES; j++)
1195              {
1196                  if( i != j)
1197                  {
1198                      if(qry_cost[i][j] < calc_qry)
1199                      {
1200                          calc_qry = qry_cost[i][j];
1201                      }
1202                   }
```

```
1203                )
1204            )
1205        )  /* End of "for" Loop */
1206
1207        /* DETERMINE UPDATE COSTS WITH NEW FILE PLACEMENT  */
1208
1209        for(i=0; i< NO_NODES; i++)
1210        {
1211            if(( i==new_node) || (solution_array[i] == i))
1212            {
1213                for(j=0; j< NO_NODES; j++)
1214                    calc_upd = calc_upd + upd_cost[j][i];
1215            }
1216        }
1217
1218        /* TOTAL COST TO RETURN IS QUERY + UPDATE        */
1219
1220        tot_cost = calc_qry + calc_upd;
1221
1222        return(tot_cost);
1223    }
1224
1225    /*****************************************************************************
1226    *  REBUILD_COST():
1227    *
1228    *    Processing: Rebuilds the query and update cost tables based upon a percent
1229    *                increase/decrease in communication costs (called if user desired
1230    *                sensitivity analysis to be performed; in which case the user supplies
1231    *                the percentage of increase/decrease desired)
1232    *
1233    *    Input:      A sign to indicate an increase or decrease is requested, and the
1234    *                percentage to be associated with this change.
1235    *
1236    *    Functions
1237    *    Called:     (None)
1238    *
1239    *    Output/
1240    *    Return:     (None)
1241    *****************************************************************************/
1242
1243    rebuild_cost(sign,percent)
1244    int sign,percent;
1245    (
1246        int i,j;
1247        if(sign == '+')
1248            svmin_cost = svmin_cost + ((svmin_cost * percent)/100);
1249        else
1250            svmin_cost = svmin_cost - ((svmin_cost * percent)/100);
```

```
1251
1252        for(i=0; i<NO_NODES; i++)
1253        {
1254            for(j=0; j<NO_NODES; j++)
1255            {
1256                if(sign == '+')
1257                {
1258                    qry_cost[i][j] = qry_cost[i][j] + ((qry_cost[i][j] * percent)/100);
1259                    upd_cost[i][j] = upd_cost[i][j] + ((upd_cost[i][j] * percent)/100);
1260                }
1261                else
1262                {
1263                    qry_cost[i][j] = qry_cost[i][j] - ((qry_cost[i][j] * percent) / 100);
1264                    upd_cost[i][j] = upd_cost[i][j] - ((upd_cost[i][j] * percent)/100);
1265                }
1266            }
1267        }
1268    }
1269
1270    /*****************************************************************************
1271    *  PR_RELTBL():
1272    *
1273    *    Processing: Prints out the contents of the relation table, which indicates
1274    *                the attribute usage per node/document for any given relation.
1275    *
1276    *    Input:      Ralation number
1277    *
1278    *    Functions
1279    *    Callad:     None
1280    *
1281    *    Output/
1282    *    Return:     print display to user
1283    *****************************************************************************/
1284
1285    pr_reltbl(relno)
1286    int relno;
1287    {
1288      int i,j,k;
1289      printf("RELNO =%d, NUMBER OF RELATIONS = %d\n",relno,attrnum[relno]);
1290      for(i=0; i<NO_ENTS; i++)
1291      {
1292       printf("\nNODE: %d ORG: %s DOC: %s \n",reltable[i].nodenum,reltable[i].orgname,
1293                        reltable[i].docname);
1294        for(j=0; j<(attrnum[relno] +1); j++)
1295        {
1296           printf("%s: ",berntable[relno].attributes[j].attrs);
1297           if(reltable[i].freq[j].part_num==NO_NODES + 1)
1298             printf(" Partition:     *KEY*    ");
```

```
1299              printf(" Partition:\t%d",reltable[i].freq[j].part_num);
1300
1301            if(reltable[i].freq[j].owner == TRUE)
1302              printf("\t%d*\n",reltable[i].freq[j].frequency);
1303            else
1304              printf("\t%d\n",reltable[i].freq[j].frequency);
1305        }
1306    }
1307)
```

Dist_Data.h

```
#define MAX_ENTRY  200  /* Max. No. of org/doc entries          */
#define DOC_SZ  30      /* Max. no. of chars in document name   */
#define DRG_SZ  30      /* Max. no of chars in organization name */
#define ATR_SZ  20      /* Max. no. of chars in attribute name */
#define MAX_ATTRS 300   /* Max. no. of attributes handled       */
#define MAX_RELS  100   /* Max. no. of relations handled        */
#define MAX_PARTS  50   /* Max. no of partitions handled        */
#define MAX_NODES  50   /* Max. no. of nodes handled            */
#define MAX_DOCS  200   /* Max. no of documents handled         */
#define TRUE    0
#define FALSE   1

   /* RELATION TABLE:  POPULATED BY NODE, DOCUMENTS and ATTRIBUTES USED */

struct REL_TABLE
{
        int nodenum;
        char orgname[DRG_SZ];
        char docname[DOC_SZ];
        int use_rel;
        int total_query;
        int total_update;
        struct FREQ
        {
                int key;
                int owner;
                int part_num;
                int frequency;
        } freq[MAX_ATTRS];
}reltable[MAX_ENTRY];

   /* BERNTABLE:  STORES BERN2 RELATIONS  */

struct BERNTABLE
{
        struct ATTRIBUTES
        {
                char attrs[ATR_SZ];
                int key;
        }attributes[MAX_ATTRS];
}barntable[MAX_RELS];
```

```
42
43
44      /* TRANSACTION TABLE:  HOLOS TOTAL NO. OF QUERY/UPOATE ACCESSES PER NOOE */
45
46    struct TRANS
47    {
48            int t_qry;
49            int t_upd;
50    }trans_tbl[MAX_NOOES];
51
52     /* TRANSACTION COST TABLES (update and query)  */
53
54    int qry_cost[MAX_NOOES][MAX_NOOES];
55    int upd_cost[MAX_NOOES][MAX_NOOES];
56
57     /* NETWORK COST TABLE  */
58
59    int costable[MAX_NOOES][MAX_NOOES];
60
61     /* SOLUTION ARRAY FOR BRANCH & BOUNO ALGORITHM */
62
63    int solution_array[MAX_NOOES];
64
65     /* GLOBAL 'HOLOER' FOR MIN_COST PARTITION/FILE PLACEMENT */
66
67    int min_cost, svmin_cost, svnode;
68
69      /* NO. OF ATTRIBUTES FOR EACH BERN2 RELATION  */
70
71    int attrnum[MAX_RELS];
72
73      /* NUMBER OF PARTITIONS CREATEO FROM BERN2 RELATION(s) */
74
75    unsigned int parts_created[MAX_NOOES];
```

DATA ALLOCATION IN A DISTRIBUTED DATABASE ENVIRONMENT

by

Kimberley Ann Johnson

B. S., Western Illinois University, 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Distributed information systems are systems which locally or geographically distribute elements within a computing system. In this environment the database designer is faced with many new design problems, one of the most critical being the distribution of data which most accurately reflects the processing needs of the organization.

The main focus of this report, is in the design of an interactive system that automates the partitioning of files and their placement onto a distributed network. A review of the relevant literature is first presented, followed by an overview of the system's design. Finally, an extensive example is provided to demonstrate the functionality and use of the interactive system.