QUERY PROCESSING OPTIMIZATION FOR DISTRIBUTED RELATIONAL DATABASE SYSTEMS : An Implementation of a Heuristic Based Algorithm

by

Moshe Stoler

Project and Report submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Systems Engineering

APPROVED:

Dr. K. Triantis
Chairman

Dr. P.M. Ghare

Dr. F. Ricci

March 1987

Blacksburg, Virginia

Table of Contents

Section	Page

1. Introduction 1
2. The Relational Database System 4
2.1 Introduction 4
2.2 Relational Operations 6
2.3 Distributed and Centralized Database Management Systems 9
2.4 Systems Engineering Considerations in the Design of the Query Optimizer 11
3. Query Optimization 14
3.1 Literature Review 14
3.2 The Heuristic Based Algorithm 17
3.3 The Linear Programming Formulation Embedded in the Algorithm 18
3.4 The Log File 22
3.5 Temporary Relation Size Estimation Techniques.. 24
3.6 An Example of the Implementation of the Heuristic Based Algorithm 28
4. Future Work 35

Table of Contents

Section .... Page

Footnotes.. 38

References....................................... 39

Appendix A Program Documentation................... 42

Appendix B....................................... 71

  B.1 User's Guide for Program Operation........... 71

  B.2 The Pascal Code............................. 75

Appendix C A Detailed Description of the Example... 114

1. Introduction

In recent years database management systems have become very important for information processing in various organizations. Due to the extensive use and rapid expansion of various information systems, there is a need for timely and cost efficient processing methods. The subsequent discussion presents one such query processing algorithm. The objective of this paper is twofold. First to discuss an implementation of a heuristic based algorithm developed by Egyhazy and Triantis [6] and which has an integer programming formulation embedded in it. Second, to suggest how this algorithm could be improved and augmented. The purpose of the Egyhazy and Triantis [6] algorithm is to find a strategy which reduces the cost of processing a query for a distributed relational database system. The implementation of this algorithm is the main contribution of this project.

The importance of this research stems from the literature on query processing for distributed database systems and from the research being conducted by both commercial and research organizations who are currently building such systems. For example, query processing for

distributed database systems has received increasing attention by a number of researchers such as Ceri et al. [1], Jacobs [2], Gavish, et al. [11] and others. Also, the designers of commercial systems such as, ORACLE [3] and INGRES [4], are currently improving their optimizers for their query processing software. These optimizers utilize query processing strategies which "optimize" or satisfy some performance criterion when answering a query. Perhaps, the use of the term optimizer is rather ambitious, since the techniques available for answering queries provide the researcher or the user only good strategies. This indicates that any claim of optimality must be carefully considered.

Section 2 begins with a brief discussion of what constitutes a distributed relational database and some of the operations which are performed on such databases in order to process a query. Section 2 continues with an explanation of the systems engineering approach used in the design of the optimizer in this project. Section 3 introduces the problem of query optimization and then presents a heuristic based algorithm (Egyhazy, et al. [6]) which finds a processing strategy in a distributed network so as to reduce the network processing and transmission costs. This approach is compared to other algorithms found in the literature. The log file, one of the data

structures used by the algorithm to estimate the resultant temporary relation size, is presented. Finally, in section 3, an example of the implementation of the algorithm is illustrated. The last section makes recommendations about future research directions. Appendix A presents the documentation of all of the Pascal procedures used to implement the heuristic based algorithm. The actual Pascal code can be found in appendix B. The detailed discussion of the example is found in appendix C.

2. The Relational Database System

2.1 Introduction

A database system is "a computer-based record keeping system: that is, a system whose overall purpose is to record and maintain information" [1] [7]. In any database system the following four major components are found: data, hardware, software, and users. The third component, the software, is the interface between the users and the data itself and is usually called the Database Management System or DBMS.

One way of storing data in a database system is by using the relational model [8]. By this we mean that data are stored in tables called relations. Each relation is comprised of a certain number of columns, called attributes, and a varying number of rows, called tuples. Each attribute has a domain, which is the set of various values that the attribute can possess. A relation that has n attributes is said to be of degree n. An example of a relation depicting a used car inventory is presented in figure 1.

No.	Car Make	Car Name	Car Model	Price
150	Datsun	280ZX	1981	8000
151	Ford	Pinto	1980	2500
152	Dodge	Omni	1983	3500

Figure 1: Used Car Inventory (UCI) relation

In this example we have a relation called UCI. It has five attributes: no., car make, car name, car model and price. This relation has 3 tuples corresponding to 3 different cars that are kept in the dealer's inventory. Note that no redundant information within a given relation is allowed, i.e, no two rows in the relation can be identical.

For all relations, tuples can be uniquely identified by referring to one or more special attributes called key attributes. In the example of figure 2 the attribute snum (serial number) is a key attribute. Each tuple can be uniquely identified by its snum value.

Snum	Last Name	Major	Grade Level	Age
123	Jones	History	Jr	21
158	Parks	Math	Gr	26
105	Anderson	Management	Sn	27
271	Smith	History	Jr	19

Figure 2: Student relation

2.2 Relational Operations

A major advantage of the relational representation lies in the ease with which relations can be manipulated to form new relations. A variety of operations have been defined to accomplish various manipulations in a relational database enviroment. The projection selection and join operations, which are used in the heuristic based algorithm, are described below.

Projection is an operation that chooses certain attributes from a relation. In other words, the projection operation picks columns out of a given relation to create a new relation. For example, consider the Student relation defined in figure 2. This relation has 4 tuples and the attributes: snum, last name, major, grade level, and age. The projection of Student on the last name and major attributes gives a new, smaller relation denoted by Student1 in figure 3.

Name	Major
Jones	History
Parks	Math
Anderson	Management
Smith	History

Figure 3: Student1 relation

Selection takes a horizontal subset of a given relation. The rows to be selected are pre-defined by a specified operand. For example, select on Student (figure 2) where major = 'Math' produces the results presented in figure 4.

Snum	Last Name	Major	Grade Level	Age
158	Parks	Math	Gr	26

Figure 4: Student2 relation

If two relations have a common attribute, they may be equi-joined over this attribute. The result is a new relation in which each row is formed by concatenating two rows, one from each relation, such that the two rows have the same value with respect to the common attribute. To illustrate this operation we introduce a new relation, the Enrollment relation in figure 5.

Snum	Class Name
123	H350
105	BA490
123	BA490

Figure 5: Enrollment relation

This relation has the attribute snum which is common to both the Enrollment and the Student relations. Suppose that we want to know the names and the class names of all of the students in the classes listed in figure 5. To obtain this we need to join the tuples from the Student relation with those tuples of the Enrollment relation over the common attribute, snum. This operation gives us the result specified in figure 6.

Snum	Last Name	Major	Level	Age	Class
123	Jones	History	Jr	21	H350
123	Jones	History	Jr	21	BA490
105	Anderson	Management	Sn	27	BA490

Figure 6: Student - Enrollment relation

After perfoming the equi-join we need to project those two attributes which are required by the query, namely last name and class name. The resulting relation is pictured in figure 7.

Last Name	Class
Jones	H350
Jones	BA490
Anderson	BA490

Figure 7: Student - Enrollment1 relation

2.3 Distributed and Centralized Database Management Systems

The term distributed database refers to a "collection of data which are distributed over different computers of a computer network"[2] [9]. In such a network, as depicted in figure 8, each site has the capability of processing local queries, and it participates in the processing of at least one global query. Data residing at remote sites needs to be accessed using communication links. The alternative to a distributed database is a centralized database in which all data are controlled and accessed by a single computer or multiple computers, and all query processing is done locally.

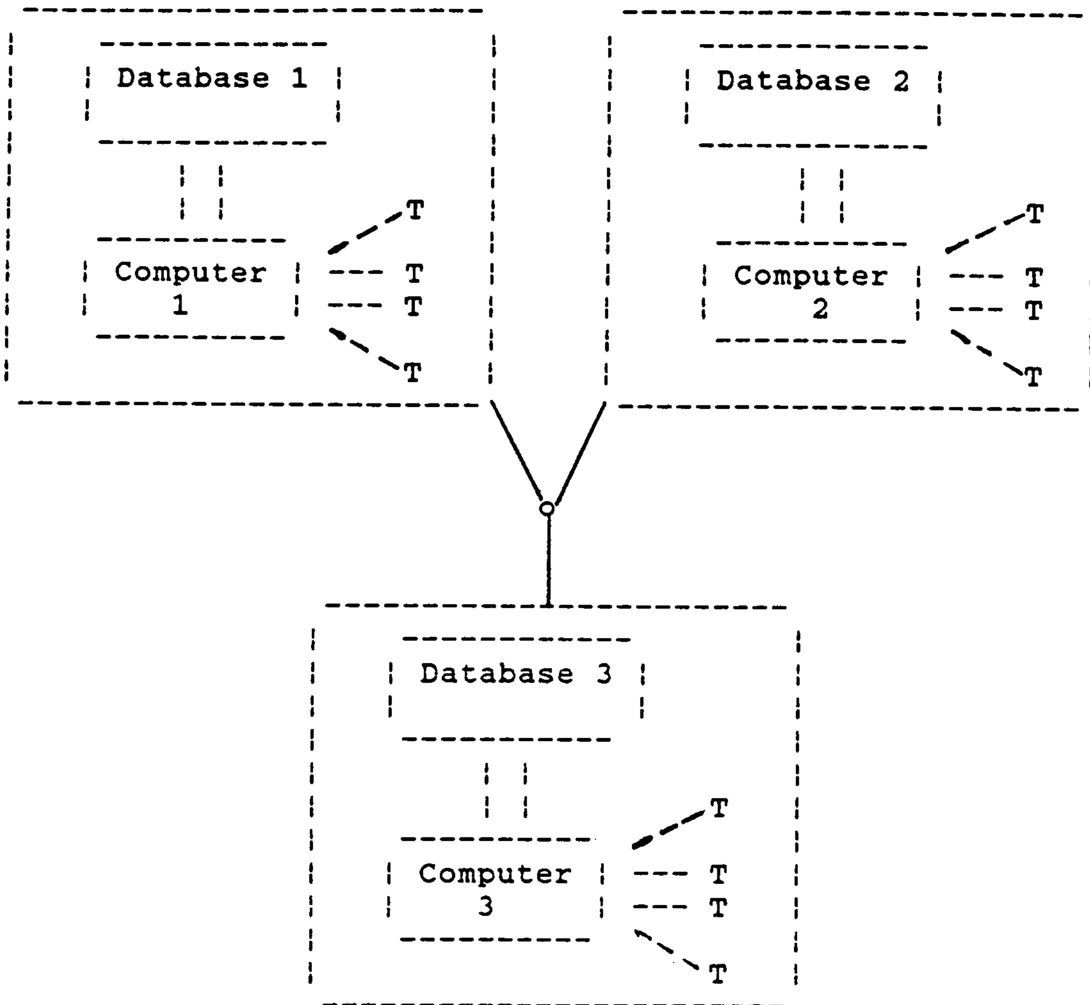

Figure 8: A distributed database.

The advantage of a distributed database originates from the fact that each site in the system stores data which is frequently accessed by the users of that particular site. In this way, there is less transmission of data from a central to a remote site, thereby reducing the transmission costs. However, in some cases there is still a need to transmit relations from one site to another. This happens

when a query originating in one site requires relations which are stored in another site in order to answer the query completely. There are other factors which make distributed systems more desirable than centralized databases. For example they facilitate incremental organizational growth, and they increase the network's overall reliability and availability [9].

2.4 Systems Engineering Considerations in the Design of the Query Optimizer

According to Systems Engineering and Analysis by B. Blanchard and W. Fabrycky, a major incentive for developing a new system is to resolve "obvious deficiencies or problems"[3] with an existing system or upon discovering such inadequacies through research [10]. This work was motivated by the demand for an efficient and economic method for handling queries, used in a variety of applications. As mentioned before in section 1, current query processing methods are no longer adequate in many cases due to the high volume of data transfer required by modern database networks.

After defining the need for a system, the following three steps in the systems engineering approach are taken: the system planning function, the system research function and the system design function [10]. The system planning

function involves the identification and evaluation of several possible solutions for a given problem. System feasibility and technical performance are two major factors that are weighed when analyzing the alternatives in order to select the preferred approach. The system research function involves both basic and applied or directed research. Finally, the systems design phase "begins with a visualization of what is required and extends through the development, test, and evaluation of an engineering or prototype model of the system"[4].Both the Egyhazy et al. [6] paper and the attached computer program follow these guidelines. This paper is devoted mostly to the presentation of the outcomes of the system research and system design functions.

The Egyhazy and Triantis [6] algorithm was selected as the basic system design method to be followed due to its simplicity and generality. This approach is flexible since it places minimal restrictions on a database. Thus, it can be easily installed in many existing database systems without changing the structure of the data or the relations, residing in the database. It must be stressed, however, that the program presented is part of a system research effort rather than a commercial product design. In this sense, the program serves as a tool for the feasibility study of the proposed algorithm [6].

The basic functions of a typical distributed database, namely, project, select and join, are identified and implemented. Other relational operations, such as the cartesian product of two relations or the difference between two relations, are ignored. This is because these three operations are sufficient to answer any query. Also, to increase system flexibility, no attempt was made to customize the program for any particular database network. Such an effort is left for later development, should specific requirements arise.

Finally, special attention is given to the interface between the users and the program. For this purpose, a way of representing relations in a database is developed. The user is required to prepare one data file only, which describes the entire database. Also, a simple method of specifying queries is developed as part of the data input phase. The program's output is a data file which contains a full description of the database after each relational operation. The order of the operations that optimize the query process is given explicitly and can be used by any database management system that will utilize this program.

3. Query Optimization

3.1 Literature Review

A query is "an expression in a suitable language which defines a portion of the data contained in the database"[5] [9]. Queries operate on exisiting relations in the database and generate, as a result, a new (temporary) relation which contains the information that answers the query. Since we are dealing with a distributed network, the problem of query optimization can be defined as determining the sequence of relational operations and their corresponding processing sites in the network so as to optimize or satisfy some performance criterion.

Most of the research effort, with respect to query optimization, studies strategies which attempt to minimize the cost of query processing by reducing the amount of data transferred among network sites. The number of query optimization algorithms, which use mathematical programming techniques to facilitate finding a query processing strategy, is limited. Two such mathematical approaches are taken by Gavish and Segev [11], and Ceri and Gottlob [1].

Gavish and Segev [11] take the approach that, instead of looking for a general solution that can be used in any database for any query type, one should try to identify special cases which occur frequently enough to justify the use of a special solution developed to enhance the

processing of recurring queries. One example of such a solution is the case of set queries which use horizontally partitioned relations. A large relation is horizontally fragmented and each fragment is then placed in a different site of the network. This is done because a typical query might require that the same operations be performed at each site on a partitioned fragment. A linear programming approach is used in order to find the best transmission strategy for the relations in the database. The linear programming function takes into consideration all the possible data transfers and chooses the least expensive strategy which leads to a solution. As mentioned earlier this solution is relevant to specific classes of queries only. Also, the case where relations need to be joined more than once in order to answer the query is not addressed. In addition, the problem of resultant temporary relation estimation is left open.

The Ceri and Gottlob [1] approach takes into consideration the specific design of some databases. In this case, one can identify a logic rule upon which relations are partitioned and distributed. If this logic rule is known to the program which transmits relations from one site to another, then some join operations which will yield empty relations can be avoided. As for the remaining relations, the order of joins is determined by a linear

programming formulation. This formulation considers the communication cost of transferring resulting relations to the site where the query originated. As in the previous algorithm, the execution is supposed to take place in two phases only. In the first phase, the maximum number of joins is performed, and in the second phase, the resulting relations are transferred to the query site for final processing. This might give rise to a less effective strategy, since a greater reduction in transmission cost can be achieved by allowing successive joins to be performed in the various sites before the results are transmitted to the site where the query originated.

The Egyhazy, et al. [6] formulation differs from the previous two approaches. First, this algorithm is mainly driven by the temporary relation size estimations, which are calculated after each operation. These calculations hinge upon the existence of statistical information. This information is stored in the log file data structure. Second, successive joins can be performed at any of the sites in the network. When a temporary relation is formed which answers the query, it is sent to the node where the query originated. However, this approach does not consider the possibility of relation fragment replication.

3.2 The Heuristic Based Algorithm

The algorithm formulated by Egyhazy et al. [6] deals with a distributed relational network with M sites. A permanent relation residing at site m is designated as f_i^m [Ap] where the subscript "i", i= 1,2...F, refers to the specific relation i and the superscript "m" to the node m. Each site can contain any number of relations with the restriction that there be no redundant relations in the database over the entire network.

The result of performing any relational operation on the original permanent relations is a temporary relation. It is assumed that temporary relations will not be sustained by the database once the query is answered. After each operation the resultant temporary relation size is calculated. This is because the temporary relation sizes are needed to be known by the integer programming model which finds the order by which relations are to be joined.

The algorithm can be briefly described as follows: When a query is issued, one can define a set of attributes specified by the query. Also known are the specific restrictions placed on designated attributes. The policy used by the algorithm is as follows:

Step 1: Project out all of the attributes specified by the query along with the key attribute(s).

Step 2: Perform all of the selections according to the

query's restrictions.

Step 3: Eliminate attributes that were required for the selection operation but are not part of the final answer to the query.

Step 4: Join the remaining relations over common attributes. The join operation is repeated until we are left with one relation which contains the desired information. The order of the joins and the site where this operation will be performed, is determined by solving a linear integer programming model which is described in the next section.

Step 5: Remove all of the key attributes from the resulting relation and send this relation to the site where the query was initiated.

3.3 The Linear Integer Programming Formulation Embedded in the Algorithm

This technique is used here to find a strategy for joining relations in the distributed network. In general, the relations to be joined reside at different sites m and n. The following notation will be used in the description of the integer programming (IP) procedure:

$$Y_{i,j}^{m,n} = \begin{cases} 1 & \text{If file } f_i^m \text{ can be joined with file } f_j^n \\ 0 & \text{otherwise} \end{cases}$$

$$Z_{i,j}^{m,n} = \begin{cases} 1 & \text{If an equi-join of } f_j^n \text{ with } f_i^m \text{ is carried out at node m} \\ 0 & \text{otherwise} \end{cases}$$

$$X_{i,j}^{m,n} = \begin{cases} 1 & \text{If file } f_i^m \text{ is sent to node n to be joined with } f_j^n \\ 0 & \text{otherwise} \end{cases}$$

$S[\ f_i^m\ ]$: Denotes the size (in bytes) of file i residing at node m.

Note that $Y_{i,j}^{m,n}$ is a parameter whose value can be determined by observing the relations' attributes. The X's and Z's are decision variables whose values, 0 or 1, are determined by solving the integer programming model.

The cost associated with processing each join is introduced via the parameters $C^{m,n}$ and CJN^m . $C^{m,n}$ is the unit transmission cost from node m to node n, whereas, CJN^m is the unit processing cost of performing a join at node m.

The objective function and the constraints of the model are as follows:

$$\underset{X^{m,n}_{i,j},\ Z^{m,n}_{i,j}}{\text{minimize}} \ \{ \ (\sum_{\substack{m,n=1\\ m=n}}^{M} \sum_{\substack{i,j=1\\ i=j}}^{F} C^{m,n} * X^{m,n}_{i,j} * S [f^{m}_{i}]) +$$

$$+ (\sum_{\substack{m,n=1\\ m=n}}^{M} \sum_{\substack{i,j=1\\ i=j}}^{F} CJN^{m} * Z^{m,n}_{i,j} * (S [f^{m}_{i}] + S [f^{n}_{j}])) \}$$

subject to:

1) $Z^{m,n}_{i,j} + Z^{n,m}_{j,i} \leq Y^{m,n}_{i,j}$ for every $m = n$, $i = j$

2) $X^{m,n}_{i,j} - Z^{n,m}_{j,i} = 0$ for every $m = n$, $i = j$

3) $$\sum_{m,n=1}^{M} (\sum_{\substack{i,j=1\\ i=j}}^{F} Z^{m,n}_{i,j} + \sum_{\substack{i,j=1\\ i=j}}^{F} Z^{n,m}_{j,i}) \leq 1, \quad i = 1, 2, \ldots F$$

$$4) \quad \sum_{m,n=1}^{M} \sum_{\substack{i,j=1 \\ i=j}}^{F} Z_{i,j}^{m,n} = 1$$

$$5) \quad X_{i,j}^{m,n} = (0,1), \quad Z_{i,j}^{m,n} = (0,1), \text{ for every } m,n,i,j$$

Constraint 1 indicates that any two joinable files can be joined only once. Constraint 2 specifies that a file f_i^m will be sent from node m to node n only if it will be joined with file f_j^n. The third constraint indicates that any file can be joined with only one other file in the network. The fourth constraint specifies that only one join can be performed in each step. Constraint 5 indicates that the decision variables $X_{i,j}^{m,n}$ and $Z_{i,j}^{m,n}$ can only contain the integer values 0 or 1.

The objective function chooses those decision variables $X_{i,j}^{m,n}$ and $Z_{i,j}^{m,n}$ which minimizes the total local processing and network transmission costs for each set of relations which need to be joined. The relations corresponding to the above X's and Z's are then joined and new relations are created. These new relations, together with the original relations which were not joined, form a new set of

relations. This set becomes the input for the IP algorithm which again selects those relations that will be joined during the next step. This process is repeated until all of the relations are joined to form a single relation which contains the required information to answer the query.

3.4 The Log File

In order to facilitate the implementation of the Egyhazy and Triantis [6] algorithm, we suggest that an additional format for describing the database be developed. This data structure consists of a condensed description of all of the relations and is called the log file. In general, one can claim that the more information we keep on each relation, the more accurate our size estimation of each relation can be. However, there could be a significant overhead involved in keeping accurate statistics. Thus, our goal is to keep as little information as possible but at the same time be able to calculate reasonable resultant size estimates.

Each relation is described by an identifying number and its location. Also, a list of the attributes and their values are kept for each relation. For key attributes, however, the list of values is omitted since, especially for large relations, this might be a long list of values that appear only once in the relation. A special

methodology for estimating the resulting temporary relation sizes after joining on key attributes is also used and is described in section 3.5.

For example, the Student relation (figure 2) is represented in the log file as follows:

file-number 1

site 1

number of tuples 4

relation size 96 bytes

([2+10+10+2] bytes * 4 tuples = 96 bytes)

Attributes:

Snum- key attribute

4 different values appear in relation, 1000 possible values exist for this attribute, size is 2 bytes

Last Name- non key attribute

size is 10 bytes

attribute values	no. of occurrences
Jones	1
Parks	1
Anderson	1
Smith	1

Major- non-key attribute

size is 10 bytes

attribute values	no. of occurrences
History	2
Math	1
Management	1

Grade Level- non key attribute

size is 2 bytes

attribute values	no. of occurrences
Jr	2
Gr	1
Sn	1

Note that even though this may appear to be a cumbersome and lengthy way of describing relations, substantial statistical information pertaining to the database is summarized. Note also that we can no longer can determine the exact contents of each individual tuple by using the information stored in the log file alone. Thus, from now on we will have to make statistical assumptions in order to estimate the resulting relation size after performing each relational operation.

3.5 Temporary Relation Size Estimation Techniques

The ability to estimate the size of a relation plays a

major role in the overall performance of this optimization technique. Incorrect size estimation methods will create inaccurate temporary size estimations which will affect the strategy chosen by the heuristic algorithm. The methods implemented in this program are based on the statistics gathered in the log file. The different ways for estimating resulting relation sizes after the performance of a project, select or join operation are briefly described subsequently.

Projection is an operation that picks columns out of a given relation to create a new relation. The new tuple size can be easily calculated after projecting on a relation by summing up the sizes of the selected attributes. Since the number of tuples remains the same, we can calculate the new relation size by multiplying the new tuple size by the number of tuples. From the above one can conclude that based on the statistics kept in the log file we can calculate the "exact" size of a projected relation. In other words, if the size estimation of the original relation is correct then the size of the projected relation will be exact.

Selection is an operation that results in a horizontal subset of the original relation. Given a predicate for the selection operation, we can calculate the exact number of tuples which satisfy this restriction for a certain

attribute. Usually, this number is smaller than the original tuple number, since some tuples are removed as a consequence of this operation. In other words, the new relation size is only a fraction of the original size. The cardinality (the number of tuples in a relation) of the temporary relation after the execution of the selection operation, can be obtained by summing the postings (number of occurrences) of the selected values. The tuple length of the temporary relation is identical to that of the original relation. Thus, the size of the temporary relation after a selection, equals its cardinality times its tuple length. Also the postings of values belonging to the other attributes will decrease in proportion to the overall reduction of the tuples in the relation.

The execution of a join operation between two relations is feasible only if these relations have a common attribute. In this case, any two rows, one from each relation, can be concatenated provided that they have the same value with respect to the common attribute. The estimation of relation size after a join has been performed is accomplished by scanning the values of the common attribute contained in the log file. This is done in parallel for both relations which are to be joined. For each value of the common attribute which appears simultaneously in both relations, the postings of the value

in the first relation are multiplied by the postings of the same value in the second relation. These partial products are finally totalled to form the new number of tuples in the relation resulting from the join. The postings of the remaining values, belonging to attributes other than the common attribute, are adjusted in the same manner used with the selection operation. In other words, the postings of each value are increased or decreased in proportion to the change in the number of tuples for each relation. As for cases involving a join over a key attribute, the new number of tuples is estimated by calculating the product of the number of tuples of the original relations, divided by the maximum number of possible values for the key attribute. For more details on these estimation procedures see Egyhazy, et al. [13].

3.6 An Example of the Implementation of the Heuristic Based Algorithm

The following example illustrates the various steps that are taken during the query optimization process.

We are given the following information: at site 1 we have relation 1; at site 2, relation 2; and at site 3, relations 3 and 4.

The relations are as follows:

f1: Supplier Relation at site 1

Attributes (sizes in bytes)	S# (2)	Sname (5)	Status (2)	Scity (6)
	S1	Smith	20	London
	S2	Jones	10	Paris
	S3	Blake	30	Paris
	S4	Clark	20	London
	S5	Adams	30	Athens

f2: Parts Relation at site 2

Attributes (sizes in bytes)	P# (2)	Pname (5)	Color (5)	Weight (2)	Pcity (6)
	P1	Nut	Red	12	London
	P2	Bolt	Green	17	Paris
	P3	Screw	Blue	17	Rome
	P4	Screw	Red	14	London
	P5	Cam	Blue	12	Paris
	P6	Cog	Red	19	London

f3: Projects Relation at site 3

Attributes (sizes in bytes)	J# (2)	Jname (8)	Jcity (6)
	J1	Sorter	Paris
	J2	Reader	Rome
	J3	Console	Athens
	J4	Collator	London
	J5	Terminal	Oslo
	J6	Tape	London

f4: Supplier-Parts-Project Relation at site 3

Attributes (sizes in bytes)	S# (2)	P# (2)	J# (2)	Qty (3)
	S1	P1	J1	200
	S1	P1	J4	700
	S2	P3	J1	400
	S2	P3	J2	200
	S2	P3	J3	200
	S2	P3	J4	500
	S2	P3	J5	600
	S2	P3	J6	400
	S2	P3	J7	800
	S2	P5	J2	100
	S3	P3	J1	200

The database network and the initial set up of the relations in the network is given by figure 9.

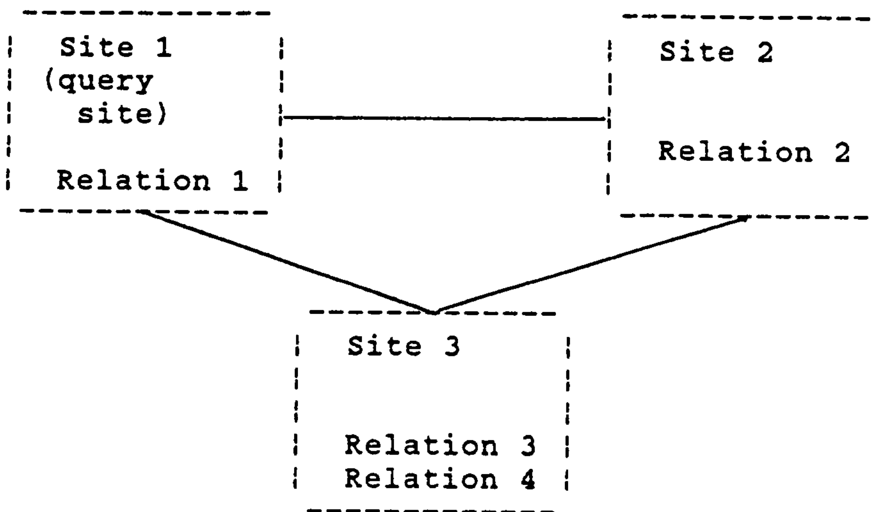

Figure 9: Initial network for the example

Assume that the transmission and local processing cost coefficients are given by the following values: $C^{1,2}=63$, $C^{1,3}=58$, $C^{2,3}=88$, $C^{2,1}=45$, $C^{3,1}=69$, $C^{3,2}=99$, $CJN^{1}=375$, $CJN^{2}=400$ and $CJN^{3}=425$.

A query originates at site 1 requesting the attributes Sname, Scity, Weight, and Quantity and the predicate relationships: Pname = Screws, Quantity $\geq$400, and Jcity = Athens.

Attributes S#, Sname, Scity, P#, Pname, Weight, J#, Jcity and Quantity (Qty), include all the attributes which are specified by the query along with key attributes. These attributes are projected out as a first step in the query processing. The relations projected f_1^1, f_2^2 and f_3^3 give rise to the new temporary relations, Tf_5^1, Tf_6^2 and Tf_7^3

repectively, relation f_4^3 remains unchanged because we need all the attributes in this relation.

After all of the projections are completed we perfom the selections required by the query. Thus we want only the tuples which satisfy the following restrictions :

Attribute #6 PNAME = SCREWS

Attribute #12 JCITY = ATHENS

Attribute #13 QTY > 400

This requires selecting on Tf_6^2, Tf_7^3 and f_4^3. Thus, after the selection we are left with temporary relations Tf_5^1, Tf_8^2, Tf_9^3 and Tf_{10}^3

At this point, the relations also contain attributes that were needed for the selections, but are not required for the final answer to the query. Those attributes are removed and we are left with attributes S#, Sname, Scity, P#, Weight, J# and Quantity.

After performing all of the possible projections and selections, the relations are ready to be joined. The 4 relations which contain relevant information are : Tf_5^1, Tf_9^3, Tf_{11}^2, and Tf_{12}^3. The linear programming method is used to select two relations out the possible four. The LP algorithm found that it is best to perform a local join at site 3 between relations Tf_9^3 and Tf_{12}^3. As a result one gets a new relation, Tf_{13}^3. The cost of this join is found to be 23800 monetary units.

Now there are 3 relations in the database, Tf^{1}_{5}, Tf^{2}_{11}, and Tf^{3}_{13}. The LP algorithm has choosen to send relation Tf^{3}_{13} to site 2 to be joined with relation Tf^{2}_{11}. As a consequence, a new relation, Tf^{2}_{14}, is created. The cost of this join is 7050 monetary units. This figure includes the cost of transfering relation Tf^{3}_{13} from site 3 to site 2 and the cost of joining this relation with relation Tf^{2}_{11}

The last join is performed between relations Tf^{2}_{14} and Tf^{1}_{5}. The LP algorithm has found that it is most economic to send relation Tf^{2}_{14} to site 1 to be joined with relation Tf^{1}_{5}. The cost of this join is 25695 monetary units. As a consequence of this join we are left with one relation in the database, relation Tf^{1}_{15}. This relation contains key attributes which are not required by the query. These attributes are now removed and a new relation, Tf^{1}_{16} is formed. This relation contains only those attributes required by the query, namely attributes Sname, Scity, Weight, and Qty. The cost of processing the query is given by the sum of the join costs, i.e., 56,545 monetary units. (For a more detailed description of the above example, refer to appendix C).

This figure can be compared with the cost of answering the same query using other methods. Another method considered by Egyhazy et al. [13] calls for the processing of queries by transmitting the smallest relation to the

site of the largest joinable relation in the network. Again, we start with relations Tf_5^1, Tf_9^3, Tf_{11}^2 and Tf_{12}^3. The first join is identical in both methods, i.e., relations Tf_9^3 and Tf_{12}^3 are joined with the cost of 23800 monetary units and relation Tf_{13}^3 is created. Next, relation Tf_{13}^3 is sent from site 3 to site 1 to be joined with relation Tf_5^1. The cost of this operation is 27813 monetary units, and the resulting relation is relation Tf_{14}^1. The final join is performed between relations Tf_{11}^2 and Tf_{14}^1. The cost of sending relation Tf_{11}^2 from site 2 to site 1 and the cost of joining these two relations is 8842 monetary units. The total cost of the joins in this case is found to be 60455 monetary units. Thus processing the query using the second method is less efficient in copmpare to the first proposed method.

A simple, conventional technique that can be used to answer this query involves sending all of the relations required by the query to the site where the query was originated. In this case, the original relations f_2^2, f_3^3 and f_4^3 are all sent to site 1. The total transmission cost is 18855 monetary units. The cost of joining the four files in site 1 is 52447. Thus, the overall cost of processing the query is 71302 monetary units. The results using the three methods are summarized in figure 10.

method	processing cost in monetary units
1	56,545
2	60,455
3	71,302

Figure 10: Summary of query processing costs

In conclusion, the Egyhazy and Triantis [6] algorithm implemented in this project did in fact derive a strategy which, when compared to other possible methods, reduced the cost of query processing in a distributed relational database system.

4. Future Work

The algorithm used here is a static decision algorithm, i.e. the order of the operations during query processing is determined in advance. The preferred order of joins is transferred to a master control program which in turn handles the remaining execution of the query.

An alternate strategy might incorporate a dynamic decision algorithm. By this we mean that the next operation (a join in this case) will not take place before the exact results of the previous operation are known to the decision maker, i.e., the IP algorithm. Thus, after each join, the postings used by the IP algorithm are updated. This will prevent a wrong decision made at an early stage from significantly influencing the optimal solution. The use of a dynamic algorithm requires additional communication costs since the new postings of the relations need to be sent back to the decision algorithm. The benefits of such a strategy, however, can be weighed against the inevitable extra communication costs. I believe that the dynamic algorithm may prove to be most beneficial when large size relations are involved, since the statistics in this case are relatively small in comparison to the size of the relation itself.

Additionally, alternative relation estimation technique could be suggested. For example, one could use the statistical information stored about the relations in order to infer the probability density function of each attribute value and the joint distribution function of that attribute. Using the probability density function, we can better estimate the expected temporary relation size as well as the variability of the expected sizes after the execution of each operation. Of course this requires some apriori knowledge about the specific data being used. This approach could decrease the amount of statistical information needed by the optimizer and consequently decrease the amount of overhead of the log file.

The existing integer programming formulation does not consider the possibility of relation fragment replication. When considering the problem of distributing data in a distributed network rules have to be found, which accomplish this distribution in a desirable way. A number of fragments may be duplicated in order to facilitate local query processing. The question then arises as to how the existing integer programming formulation can be augmented to take into account replicated fragments.

Lastly, this algorithm considers only local processing and transmission costs. This approach does not explicitly take into account user response time. One would have to

understand the relationship between the user response time and the query processing cost to be able to enhance the existing approach to also include the modelling of the user response time.

Footnotes

1. C. Date, An Introduction to Database Systems. Addison-Wesley, Menlo Park, California, p. 3, (1982).
2. S. Ceri and G. Pelagatti, Distributed Databases: Principles and Systems. McGraw-Hill, New York, p. 6, (1984).
3. B. Blanchard and W. Fabrycky, Systems Engineering and Analysis. Prentice-Hall, Inc., New Jersey, p. 20, (1981).
4. B. Blanchard and W. Fabrycky, Systems Engineering and Analysis. Prentice-Hall, Inc., New Jersey, p. 27, (1981).
5. S. Ceri and G. Pelagatti, Distributed Databases: Principles and Systems. McGraw-Hill, New York, p. 26, (1984).

References

1. S. Ceri and G. Gottlob, "Optimizing Joins between Two Partitioned Relations in Distributed Databases", Journal of Parallel and Distributed Computing, 3, 183-205 (1986).

2. B. Jacobs, Applied Database Logic, Volume I, Fundumental Database Issues, Prentice-Hall, Inc., (1985).

3. ORACLE Corporation, UFI Terminal User's Guide, Relational Software Incorporated, Menlo Park, California.

4. M. Stonebraker and P. Rubenstein, "The INGRES Protection System", Proceedings of the Association for Computing Machinery Conference, PP. 81-84, Association for Computing Machinery, New York, October 1976.

5. B. Jacobs, Applied Database Logic II, Heterogeneous Distributed Query Processing, Prentice-Hall, Inc., 1987, (forthcoming).

6. C. Egyhazy and K. Triantis, "An Integer Programming Formulation Embedded in an Algorithm for Query Processing Optimization in Distributed Relational

Data Base Systems", Computer and Operations Research, 1987, (forthcoming).

7. C. Date, An Introduction to Database Systems. Addison-Wesley, Menlo Park, California (1982).

8. C. Codd. "A Relational Model for Large Shared Data Banks", Communication of ACM, 13, 6 (June), 377-387, 1970.

9. S. Ceri and G. Pelagatti, Distributed Databases: Principles and Systems. McGraw-Hill, New York, (1984).

10. B. Blanchard and W. Fabrycky, Systems Engineering and Analysis. Prentice-Hall, Inc., New Jersey, (1981).

11. B. Gavish and A. Segev, "Query Optimization in Distributed Computer Systems", Management of Distributed Data Processing, North-Holland Publishing Company, 1982.

12. R. Epstein, Query Processing Techniques for Distributed Relational Database Systems. UMI Research Press, Michigan (1982).

13. C. Egyhazy and K. Triantis, "A Query Processing Algorithm for Distributed Relational Data Base Systems", The Computer Journal, 1987, (forthcoming).

14. D. Kroenke, Database Processing. SRA, Chicago (1983).

15. M. Syslo and M. Deo, Discrete Optimization Algorithms with Pascal Programs. Prentice-Hall, Inc., Englewood Cliffs, New Jersy, 1983.

APPENDIX A

A.1 Program Documentation

SCOPE

The following document describes each of the modules which comprise the Query Processing Program. Each module is described in terms of its function, its relation to other modules, and the input / output parameters required by this module.

(* In this document the words file and relation are used interchangeably.)

DATA STRUCTURES USED BY HEURISTIC BASED ALGORITHM

In order to facilitate the understanding of the program, a description of the each data structure and its use is given in the following section:

FILE MATRIX STRUCTURE (FILE-MTX)

This is an array which contains information on the relations in the database. Each line in this array corresponds to one relation and has the following columns:

File #: A number associated with a relation or the relation's name.

File site: A number which defines where this relation resides, i.e. the relation's location in the distributed network.

Active: A boolean variable which indicates if a relation can be manipulated any further (an active relation) or if it has been already manipulated.

A1...An: This group of columns defines the

attributes which a certain relation contains. For example, if we have A1=0 and A2=1, this means that attribute 1 doesn't appear in relation 1 but attribute 2 is part of this relation.

Total Tuples: This variable indicates the number of tuples in this relation.

Toatal size: A number indicating the size of the relation in bytes. To calculate this value one should sum up the size of all of the different attributes in the relation and multiply this by the number of tuples.

PR1 : A variable indicating the original relation from which this relation was created. For example, if relation 1 is projected or selected to give rise to relation 6 then we say that relation 1 is the parent of relation 6.

PR2: This is similiar to PR1 in the case where a join operation is involved. For example if relations 5 and 6 are joined to give rise to relation 7 then we say that relation 5 is the parent1 and relation 6 is the parent2 of relation 7.

FILE #	SITE	ACTIVE	Ai	TOTAL TUPLE	TOTAL SIZE	PR1	PR2
1	1	TRUE	0	12	132	0	0
2	1	FALSE	1	15	105	0	0
3	2	TRUE	1	5	35	2	0
4	3	FALSE	0	8	88	0	0

Figure A.1: The File Matrix

LOG FILE STRUCTURE

The log file is an array of linked-lists as shown in figure A.2 below. Each linked-list corresponds to a different relation. The pointers to these lists are stored in a pointer vector called HEADS, i.e. HEADS [1] points to the posting structure of relation 1.

Each column in the posting structure is a linked list which corresponds to one attribute in the relation.

Each attribute in the linked list is described by the following fields:

Down: A pointer which points to a value associated with this attribute.

Attribute Type: Distinguishes between attributes which are of alphanumeric type and those which are numeric data. If Type=1 then the information is alphanumeric, e.g. red,green etc.

Key Attribute: This field distinguishes between attributes which are key attributes and those which are regular attributes.

Attribute Number: Indicates the number of an attribute.

Value Range: Specifies the maximum number of values for a particular attribute. This is used

	mainly to estimate the postings of the key attributes.
Actual Values:	The actual number of different values for a particular attribute in that specific relation.
Attribute Size:	The size,in bytes, of an attribute.
Next Attribute:	A pointer to the next attribute in the relation.

The down field in the above description points to another linked list, the value list, which has the following fields:

Down:	A pointer to the next value in the list.
Occurrences:	Specifies how many times this value appears in the relation.
Avalue:	Stores alphanumeric values in case that this attribute is of alphanumeric type.
Nvalue:	Stores numeric values in case this attribute is of numeric type.

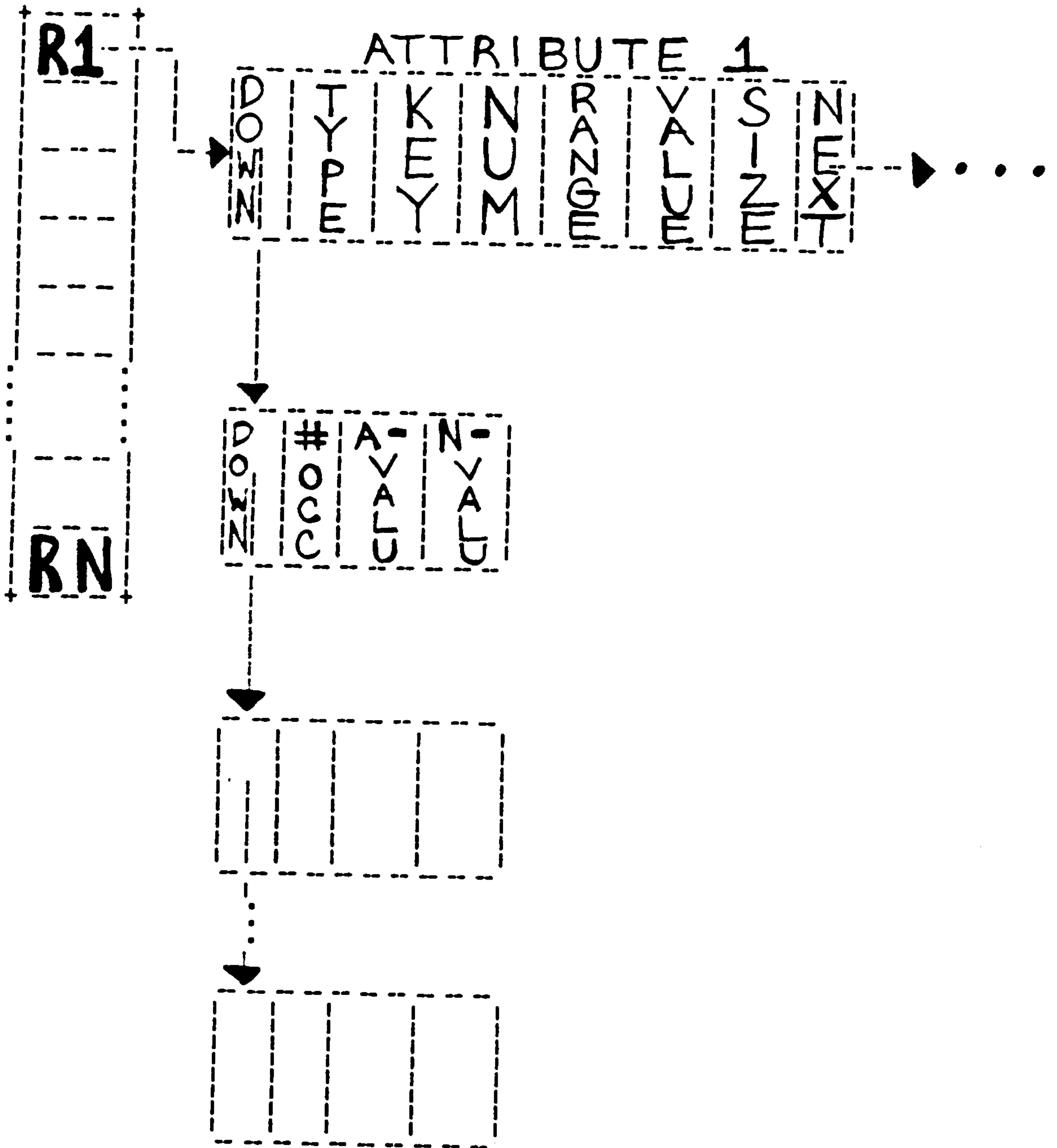

Figure A.2: The Log File.

QUERY MATRIX (QRY MTX)

The query in this program is read from a data file and is represented internally as an array data type. The query array is comprised of the following columns:

Attribute #: The number of the required attribute specified by the query.

Attribute Type: Describes the nature of an attribute; if the value in this field is 0 then the attribute holds numeric values otherwise the values are alphanumeric.

Must: A boolean variable indicating whether this attribute should appear in the final response to a query. If 1, then we want this attribute as part of the relation which answers the query.

Operator: An arithmetic operation associated with this attribute, GT - greater than or equal to, LT - less than or equal to, EQ - equal, ALL - no restriction on the values.

Avalue: When dealing with an alphanumeric type attribute this field holds the alphanumeric predicate of the query.

Nvalue: Same as Avlue when a numeric value is

involved.

ATT	ATT_TYPE	MUST	OPERATOR	AVALUE	NVALUE
2	1	1	ALL		
4	1	1	ALL		
8	0	1	All		
13	0	0	GT		400
12	1	0	EQ	PARIS	

Figure A.3: The Query Matrix

The last line in figure A.3, calls for those tuples in the database which contain attribute 12 and the value of this attribute is Paris. However, this attribute is not required as part of the final answer to the query.

COST OF JOIN

This is a vector which contains the local proccessing cost per byte for joining two files at various sites in the network.

site 1	site 2	site 3	site 4
375	400	425	250

Figure A.4: The Cost of Join vector

For example, in figure A.4, the local processing cost

at site 2 is 400 monetary units per 1 joined byte.

COST OF TRANSMISSION (TX COST)

This is an array which holds information on inter-site transmission costs.

site	1	2	3	4
1	0	63	58	70
2	45	0	88	72
3	69	99	0	65
4	15	90	75	0

Figure A.5: Cost of Transmission Matrix

For example, in figure A.5, the transmission cost between site 2 and site 3 is 88 units per transmitted byte.

THE JOIN MATRIX

The join matrix is an array which contains pairs of relations which have at least one common value in respect to some common attribute. Each column in the array defines one such pair. The first two rows specify the relations'

sites while the 3rd and the 4th rows denote the relations' numbers. The common attribute, upon which these two relations are joinable, is stored in the 5th row. For example, in figure A.6, the third row defines two joinable relations. Relation 9 at site 3 is joinable with relation 11 at site 2. The attribute which is common to both relations is attribute 5.

Site m ->	1	3	3	2
Site n ->	3	1	2	3
Relation i ->	5	9	9	11
Relation j ->	9	5	11	9
Common Attribute ->	1	1	5	5

Figure A.6: The Join Matrix

THE LINEAR PROGRAMMING MATRIX (IP MTX)

The linear programming matrix is an array which contains the information on the LP function and the constraints. It is best illustrated by the following example:

minimize 13 X1 + 15 X2 + 14 X3 + 11 X4

subject to -4 X1 - 5 X2 - 3 X3 - 6 X4 =< - 96

-20 X1 - 21 X2 - 17 X3 - 12 X4 =< -200

-11 X1 - 12 X2 - 12 X3 - 7 X4 =< -101

where X1, X2, X3, X4 >= 0 and integer.

Figure A.7 gives the internal representation of the function and its constraints in the computer's memory.

13	15	14	11	0
-4	-5	-3	-6	-96
-20	-21	-17	-12	-200
-11	-12	-12	-7	-101
-1	0	0	0	0
0	-1	0	0	0
0	0	-1	0	0
0	0	0	-1	0

Figure A.7: The Linear Programming Matrix

After using the LP algorithm, the solution obtained is:

The minimized function: -A[0,5] = 187
X1 = A[4,5] = 0
X2 = A[5,5] = 0
X3 = A[6,5] = 0
X4 = A[7,5] = 17

OVERVIEW

The first step of the program is to input the statistical information concerning the relations of the

database. This information is stored in the log file and the file matrix data structures. Next, the query itself is read and stored in an array called the query matrix. The program examines the various fields of this matrix and decides which relations in the database are necessary to answer the query. For these relations it determines those attributes which should be eliminated and those which should be preserved for further processing. The key attributes are identified and are projected along with the other attributes. After the initial projection is completed the sizes of the new temporary relations are evaluated and stored in the appropriate fields of the file matrix structure. The program then examines that part of the query which contains the various restrictions on the attributes. The values of the attributes are sorted and those values which do not match the restrictions are eliminated from the log file. Again, the sizes of the new relations are estimated according to the method described by Egyhazy et al. [6]. A second projection is performed to eliminate attributes which were required by the selection phase but are not part of the final answer to the query.

The remaining relations are those relations which need to be joined to form a relation with the required information. In order to decide upon which relations to join, a special table, the join matrix, is created. This

table contains pairs of relations which have common attributes and common values and therefore are joinable. The LP algorithm is used to determine the least expensive join out of all the possible joins. This process is repeated until all of the relations are joined to form a single relation which answers the query. As in the case of projection and selection the size of the temporary relations after each join is estimated. As a last step, we remove the key attributes which helped in joining the files but are not part of the answer to the query.

The following section contains the description of all of the procedures used in this program.

Procedure: READ-DATA-FILE

Functions: 1. Creates the log file which is an image of the original database, in a linked list form.
2. Creates an auxillary data array called File Matrix.

Called By: MAIN

Call : None

Input : 1. File_Structure input files (db_map.dat)

Output : 1. FILE_MTX, a matrix which contains information on all the files in the original structure. (see fig A.1)
2. LAST_FILE, a variable which indicates the number of files (or lines) in the matrix.

Procedure: PRINT_DB

Functions: 1. Prints out a description of the database. This is usualy done after reading the original database or after performing a relational operation on the database.

Called By: MAIN

Call : None

Input : 1. The log file data structure.
2. File Matrix.

Output : 1. Text file named DB_OUT.DAT.

Procedure: READ_QRY

Functions: 1. Reads a text file which contains the required query.

Called By: MAIN

Call : None

Input : 1. Qry_Map.Dat - a text file containing the query description.
2. File Matrix.

Output : 1. Qry_Map - an array which contains the required query.

Procedure : CALCULATE_SIZE

Functions: 1. This procedure scans the linked list of a file in the log file and calculates the size of this file.(A file size is estimated as the sum of the atributes' size in a tuple times the number of tuples in that file.)

Called By: MAIN,DELETE_ATT

Call : None

Input : 1. File name, log file.

Output : 1. The new value for the file's size is stored in the corresponding row of the File Matrix.

Procedure: PREPARE_INITIAL_PROJECTION

Functions : 1. This procedure reads the Qry_Map array and prepares three lists:

i) A list with all of the attributes which appear in the query.(These attributes are to be projected in the first step of the query processing.) The name of this list is Project_Initial_List.

ii) A list of attrirbutes that should be selected. (for example when the query requires: att1 > 5)
This list is stored in an array-like list called: Select_Matrix.

iii) A list of attributes which appear in the final answer to the query.
The name of this list is Project_Must

Called By: MAIN

Call : None

Input : 1. Qry_Mtx array.

Output : 1. The following lists: Project_initial_list,
Project_Must,
Select_Matrix

Procedure: CHECK_IF_NEED_TO_PROJECT

Functions: 1. This procedure checks if a certain file needs to be projected, in other words, if this file has any of the attributes which appear in the Project_Initial_List.

Called By: PROJECT

Call : None

Input : 1. Project_Initial_list.
2. File name. (indicates the file that is examined)

Output : 1. Proj_Yes_No, a boolean variable to indicate whether a projection should be performed on this file.

Procedure: DISCONNECT_ATT

Functions: 1. This procedure is used to delete attributes which are not required by the query. It traverses the linked list in the log file for a particular file and eliminates (disconnects) unrequired attributes.

Called By: MAIN

Call : None

Input : 1. Heads[old file], Heads [new file]: points to the linked list of the original file and to the new (updated) file.
2. Delete_list: a list which contains the names of the attributes which are to be disconnected from the list.

Output : 1. Updated log file.

Procedure: DELETE_ATT

Functions : 1. This procedure creates a new line in the File Matrix array. This line corresponds to a new file created by the program after a projection on a certain file took place.

2. The procedure prepares a list of attributes that should be deleted from the log file.

Called By: PROJECT

Call : Disconnect_Att
Calculate_Size { of a file}

Input : 1. Proj_List: a list containing the attributes to be projected.

Output : 1. A new line in the file matrix.

2. Delete_List: a list of unrequired attributes.

Procedure : PROJECT

Functions: Reduces the original files to files containing only those attributes which are required by the query, eliminates unrequired files.

Called by: MAIN

Calls : Check_If_Need_To_Project and Delete_Att.

Input : Proj_list: a list containing the required attributes.

Output : Updated FILE_MATRIX, updated log file.

Procedure: SELECT_ALL_FILES

Functions: 1. This procedure calls various routines to select upon the files in the database as required by a query.

Called By: MAIN

Call : Check_If_Need_To_Select, Select_File_Upon_Att New_Line,Count_Tuple_Num, Update_Value_Occurrences, Calculate_Size.

Input : 1. The log file.

Output : 1. Updated log file.

Procedure: SELECT_FILE_UPON_ATT

Functions: 1. This procedure traverses the log file, finds the required attribute and eliminates the values which are not required by the query.

Called By: Select_All_Files

Call : Skip_value,Disconnect_Value

Input : 1. File number: the file to be selected.
2. Att_To_Select: the attribute to be selected.
3. Select_mtx_line: A record which specifies the required selection (for example: Att1 = 7)

Output : 1. An updated log file.

Principles of operation:
Given the file for selection, an attribute and a predicate of this attribute (e.g. A3 = 5), SELECT will use the file number to access its posting in the log file.
SELECT scans the posting list for this attribute, value by value. If a value is equal to the given predicate, its posting is added to a counter which counts the total number of tuples.

Procedure: NEW_LINE

Functions: 1. This procedure creates a new line in the File Matrix array after a selection was performed. The new line is identical to the old line except for the new file number, associated with this line. The procedure deactivates the old line in the File matrix by setting the value of File_Mtx[i].active to "False".

Called By: Select_All_Files

Call : None

Input : 1. File name.

Output : 1. A new line in the File Matrix.

Procedure: UPDATE_VALUE_OCCURRENCES

Functions : 1. This procedure updates the postings associated with the attributes which were not selected. The procedure uses the adjusting factor which was calculated in Count_Tuple_Num procedure. The posting of each value (of the attributes which were not selected) is multiplied by the adjusting factor to get the estimated posting of the other values in this file.

Called By: Select_All_Files

Call : None

Input : 1. File number,adjusting factor

Output : 1. Updated posting values.

Procedure : COUNT_TUPLE_NUM

Functions: 1. This procedure counts the remaining tuples within an attribute after a selection upon this attribute was performed.

2. Also, the procedure calculates a scaling factor which is later used to update the postings of the values of the other attributes in this file. This factor is the division of the new tuple count with the old tuple count.

Called By: Select_All_Files

Call : None

Input : 1. File number,attribute number

Output : 1. Adjust_factor: a variable used to adjust the postings in the remaining files.

Procedure : DISCONNECT_VALUE

Functions: 1. This procedures eliminates (disconnects) a value from the linked list.

Called By: Select_File_Upon_Att

Call : None

Input : 1. Current_Value: a pointer pointing to a value of one of the attributes.

Output : 1. The pointer Current_Value is updated.

Procedure : SKIP_VALUE

Functions: 1. This procedure is used while traversing the linked list. It causes the pointer of this list to skip on the current member in the list and to point to the next value.

Called By: Select_File_Upon_Att

Call : None

Input : 1. Current_Value: a pointer to a value in the linked list.

Output : 1. The pointer Current_Value is incremented and points to the next value.

Procedure: CHECK_IF_NEED_TO_SELECT

Functions: 1. This procedures checks if a file contains a certain attribute (to be selected later on).

Called By: Select_All_Files

Call : None

Input : 1. File_Number: Indicates the file to be checked.
2. Att_To_Select: Indicates the required attribute.

Output : 1. Select_Yes_No: a boolean variable indicating whether this file is to be selected.

Procedure : NEW_LINE AFTER JOIN

Functions: 1. This procedure creates a new line in the File Matrix array after a join operation was performed. The new line contains the attribute list for the joined file, the location of the joined files,the parents of the joined file and the new file number. The procedure deactivates the old lines in the File matrix by setting the value of File_Mtx[i].active to "False".

Called By: Join Two Files

Call : None

Input : 1. File numbers of the two joined files .

Output : 1. A new line in the File Matrix.

Procedure: CHECK IF COMMON ATTRIBUTE

Functions: 1. This procedure gets as an input two file numbers. It scans the corresponding lines in the File Matrix and looks for a common attribute between the two files.

Called By: Check IF Joinable.

Call : None

Input : 1. The names of two files.

Output : 1. The number of a common attribute. If there is none, the procedure will return the value 0.

Procedure: CHECK IF COMMON VALUE

Functions : 1. This procedure gets the names of two files and checks for common values within a given attribute. If there are such values (which appear in both files) then the two files are joinable.

Called By: Check If Joinable

Call : None

Input : 1. Names of two files and a common attribute.

Output : 1. A boolean variable 'Joinable Yes No' which indicates whether these files are joinable or not.

Procedure : Check If Joinable

Functions: 1. This procedure activates other procedures in order to determine if a pair of files are joinable.

Called By: Create Join Matrix

Call : 1. Check If common attribute.
2. Check if common value.

Input : 1. The names of two files.

Output : 1. A boolean variable indicating if these files are joinable.

2. The number of the common attribute, if there is one.

Procedure: CREATE JOIN MATRIX

Functions: 1. The Join Matrix is a data structure which contains information on all of the joinable files and their current locations. This procedure creates this structure by examining all of the active files in the database and determinig if these files are joinable or not. For every pair of joinable files a column in the Join Matrix array is created . This column contains the names of the joinable files,their locations and the common attribute.

Called By: Determine Join

Call : None

Input : 1. The global array File Matrix.

Output : 1. The global array Join Matrix.

PROCEDURE: JOIN TWO FILES

Functions: This procedure takes two files and joins them in respect to a given common attribute.

Called by: Activate Join

Calls : Skip Value, Disconnect Value, Count Tuple Num, Update Value Occurrences, Calculate Size, New Line After Join.

Input : Names of two files to be joined and the common attribute.

Output : 1. An updated log file.
2. New calculated number of occurrences for the values within the common attribute.
3. A new line in the File Matrix.

Principles of operation:

JOIN sets up pointers to the posting lists of common attribute in the two given files and scans these lists in parallel. Whenever common values are found, the product of their posting is calculated and added to a counter which counts the new number of tuples.

Procedure: CREATE LP PROCEDURE

Functions: 1. In order to determine which files should be joined, a linear programming method is used. This procedure creates the linear programming (LP) function and the constraints. A special array, LP_MATRIX, which contains the above information is created.

Called By: Determine Join

Call : None

Input : 1. Join Matrix array.

Output : 1. An array called LP_MTX which contains the linear programming function and the related constraints for this function.

GProcedure: ALLINTEGER

Functions: 1. This procedure solves the following integer LP model:

minimize $a X$

subject to $B X < c$

(for more details refer to: Descrete Optimization Algorithm, M.M. Syslo, Prentice-Hall, 1983)

The procedure is used to determine which files are to be joined in order to minimize local processing and transmission costs.

Called By: Determine Join

Call : None

Input : 1. N number of variables in the LP function.

M n + the number of constraints.

IP_MTX array containing the function and constraints.

Output : 1. The n+1 column of the IP_MTX array contains the solution to the LP function.

GProcedure: DETERMINE JOIN

Functions: 1. Calls a procedure which builds the LP matrix.

2. Calls the procedure Allinteger which solves the LP model.

Called By: MAIN

Call : Create Lp Mtx, Allinteger

Input : None

Output : The solution vector of the LP model.

GPROCEDURE: ACTIVATE JOIN

Functions: The results of the LP procedure (i.e. the Zi's which are set to '1' by the LP procedure) are kept in the global array IP_MTX. The procedure retrieves the value of the Zi's and calls a procedure to perform the joint if necessary.

Called by: MAIN

Calls : Join Two Files

Input : IP_MTX array.

Output : New joined files.

A P P E N D I X B

B.1 User's Guide for Program Operation

The following section contains directions for running the Query Optimization program.

This program runs on the VAX 11/780. To log on the computer and run the program, follow the instrucions below:

1. [nva] Enter Destination: Respond: lan <ret>
   # Respond: call 350 <ret>
   CALL COMPLETED TO (XXX<X) Respond: <ret>

2. USERNAME: Respond: username <ret>
   PASSWORD: Respond: password <ret>
   Welcome to VAX/VMS version #
   $
   /you are now logged on the VAX/

3. Using the editor, create the following input data files:

   DB_MAP.DAT: A description of the relations in the distributed database.

   QRY_MAP.DAT: A description of the required query,

   COST_OF_JOIN: Contains the prices for joining two relations in the various sites.

   COST_OF_TX: Contains the prices for transmitting a relation from one site to the other.

4. To compile and run the program:

   $ pascal opt (opt.pas is the source code of the optimizer program)

   $ link opt

   $ run opt

5. The output of the program can be found in file DB_OUT.DAT

An example of the DB_MAP.DAT file is shown below:

```
% 1    1    5
% 1    s#            1  1  5   5   2
% s#                 5
/
% 2    sname         1  0  5   5   5
% smith              1
% jones              1
% blake              1
% clark              1
% adams              1
/
% 3    status        0  0  4   3   2
% 20                 2
% 10                 1
% 30                 2
/
% 4    scity         1  0  5   3   6
% london             2
% paris              2
% athens             1
/
*
$
```

One should note the following format rules which were used when composing this file:

Note 1: Column 1 contains either *, /, or %.

* -> end of file.
/ -> end of attribute description.
% -> attribute description follows on same line.

Note 2: The figures in Row 1 signify that relation 1 stored at site 1 contains 5 tuples.

Note 3: The first two figures in Rows 2, 5, 12 and 17 signify the attribute number and the attribute name.

Note 4: The following five figures in Rows 2, 5, 12 and 17 indicate:

figure 3 = 1 -> alphanumeric field
0 -> numeric field.

figure 4 = 1 -> key attribute.
           0 -> non-key attribute.

figure 5 = number of different values for the attributes which appear in the relation.

figure 6 = the maximum number of possible values for the attribute which may appear in the relation.

figure 7 = the attribute size in bytes.

Below is a sample of the QRY_MAP.DAT file. This file contains an encoded representation of the query used in the example presented in section 3.6 of this report:

Query: request attributes Sname, Scity, Weight, and Quantity select those tuples for which Pname = screw, Quantity $\geq$ 400, and Jcity = athens.

QRY-MAP.DAT file:

```
 2   1   1          all
 4   1   1          all
 8   0   1          all
 6   1   0          eq screw
13   0   1          gt 400
12   1   0          eq athens
9999
```

This file also follows specific format rules:

Note 1: Column 1 contains attribute numbers or 9999 to indicate the end of a query.

Note 2: Column 2 indicates the type of an attribute:

1 -> alphanumeric field.
0 -> numeric field.

Note 3: Column 3 indicates:

1 -> attribute was requested to appear in the answer to the query.
0 -> attribute is only used in the predicate of the query and was not requested to appear in the answer to the query.

Note 4: Column 4 indicates which values of the attribute are to be selected by the query.

all -> all of the possible values of the attribute are required, i.e. a projection is performed over the attribute.

eq <value> -> select those tuples for which the attribute value equals <value>.

gt <value> -> select those tuples for which the the attribute value is greater than or equal to <value>.

lt <value> -> select those tuples for which the the attribute value is less than or equal to <value>.

B.2 The Pascal Code

```
 Program dbase(input,output,db_map,db_out,qry_map,
            cost_of_join,cost_of_tx,lp, result,datain);
  {  This program is used in a DATA BASE  enviroment.
  Purpuse: given a QUERY, the purpuse of this program is to
  determine which files should be sent to which sites in
  order to accomplish the required query The operations
  performed by this program are projection ,selection and
  join. The program first performs the selection and
  projection operation on every file in the DBMS,then the
  program performs the join operation between files
  residing in the same site. The last stage is to perform
  an inter-site join between files residing in different
  sites. The rule which governs this operation is: send the
  smallest file to be joint with the largest file (if
  possible ).As the program proceeds , files sizes are
  calculated or estimated, This values are used as a key
  parameter which determines, during the final joins,which
  files should be transmitted to which site.

  The input to this program is 1 data file which represents
  the files in the data base system.( this file is not  a
  copy of the original files ,it includes the neccessary
  data needed to perform a file size estimation after each
  project ,select or a join operation. ) }

  { each file in the DB is represented by one matrixline. }
  type matrixline = record
     file_num : integer;
     { actually a number associated whith this file }
     file_site : integer;
      { for example 1,2,17 }
     active    : boolean;
      { true if file is active ,false if not }
     att_list : array [1..20] of integer;
     { 1 if the file contains this
                                    attribute, 0 if not }
     tuples_num : real;
      { how many lines ,or tuples, in this file }
     file_size : real;
      { total size,in bytes, for this file }
     parent1  : integer;
      { (history) what is the origin of this file }
     parent2  : integer;
     end; {matrixline}
     type  arrmn1 = array[0..100,1..100] of integer;
```

```
type names = packed array[1..10] of char ;
     {represents alphanumerical values}
type value_pointer = ^value_entry ;
  value_entry = record
         down : value_pointer ;
         { points to the next value }
         occurrences : real ;
         {number of occurrences for this value}
         avalue : packed array[1..17] of char;
         {alphanumeric data}
         nvalue : integer ;{numeric data}
 {note : an attribute can have numerical values 12, 135 etc.
    or alphanumerical values like jones, blue,london etc.}
         end;

 type attribpointer = ^attrib_head ;
                attrib_head = record
                down : value_pointer;
                { points to the first value }
                att_type  : integer ;
                {if 1 then data is alphanumeric}
                key_att : integer ;
                { 1 if this a key attribute }
                att_num : integer ; { attribute number }
                att_name : packed array[1..12] of char ;
                {attribute name }
                value_range: integer ;
                {the maximum possible different
                values for a particular attribute }
                actual_values: integer ;
                {the actual number of different
                values for a particular attribute }
                att_size : integer ;
                { size of an attribute in bytes}
                next_att : attribpointer ;
                {points to next attrib_head}
                end ; {of record}
 type intlist = array[1..20] of integer ;

 { a query is stored in a matrix like form, each line
 in the query corresponds to an attribute. The matrix as a
 whole represents a set of required attributes and values
 associated  with these attributes

 att#  ,   att_type , must ?  , < > =  , value

 att# : attribute number
 att_type : 0 for a numeric attribute ,
                             1 for an alphanumeric att.
```

```
 must ? :  is this attribute required in the final answer
           to the query ? ; 0 for no ,  1 for yes
 < > =  : arithmetic operation associated with this
          attribute.
value : a value upon which we want to select
example : select upon jcity = athens ,
        ( jcity is attribute # 12 in the query matrix this
        will appear as:
             12   1   0   =   athens
}

type qryline = record
     att_num : integer ;
     att_type :integer ; { 0 numeric ,  1 alphanumeric }
     must     :integer ; { 0 no     , 1 yes       }
     operator : ( gt, lt,eq ,all);
             { gt greater than, lt less than,
                              eq equal, all no restriction}
     avalue : packed array[1..10] of char;
     {alphanumeric data}
     nvalue : integer ;{numeric data}
 end;  { of record qryline}

 var ii,mm,nn,count1,kmax : integer;
 var last_file : integer;
 var no_solution : boolean;
 var end_join : boolean ;
 var price : array[1..20] of integer;
 var file_mtx  : array[1..20] of matrixline ;
 var qry_mtx   : array[1..20] of qryline ;
 var select_mtx : array[1..20] of qryline ;
 var db_map, db_out,qry_map,lp,result,datain : text ;
 var cost_of_join,cost_of_tx : text;
 var heads :array[1..20] of attribpointer ; {vector of
                                   file-head pointers}
 var initial_project_list : array [1..20] of integer;
 var project_after_select : array [1..20] of integer;
 var project_final : array[1..20] of integer;
 var key_list : array [1..20] of integer;
 var test : array [1..20] of integer;
 var query_attrib_list : array[1..15] of integer ;
                              newfile : boolean ;
 var newline : matrixline ;
 var proj_list ,delete_list : intlist;
 var qrlist : intlist ;
 var join_mtx : array[1..5,1..20] of integer ;
 var join_cost : array[1..5] of integer;
 var tx_cost : array[1..5,1..5] of integer;
 var ip_mtx : arrmn1;
```

```
{**********************************************************}

procedure read_data_file(var db_map :text);
{ this procedure reads a description of the data base. the
follwing format is used for representing the files in the
data base:

file# , file_site , total_tupples
attr# , key , attr-name , attr-type ,

              # of possible , # of actual  , atttr.
                  values          values      size

value a , # of occurrences
value b , # of occurrences
.
.
.
value z , # of occurrences
/         comment  end of attribute description
.
*         comment  end of file description.
.
$         comment  end of data base description or end of
                                               input file.

}
var i,j :integer;
var x, last_att,current_att : attribpointer ;
var y, last_value,current_value : value_pointer ;
var test_char : char ; { one of the following : /,*,$,% }

begin
reset (db_map); { move pointer to the begining of the file}

for j:= 1 to 20 do  { reset the list containing the key
                                                attributes}
    key_list[j] := 0;

i := 0;       { counter for the file_mtx array }
read (db_map,test_char);
while test_char <> '$' do  { continue as long as not end of
                                               input file }
begin

 i := i+1;    {increment counter by 1}
 last_file := i;  { this variable denote the last line in
                                           the matrix_file}
```

```
with file_mtx[i] do
begin
          {  fill in 1 line in the file_mtx}
 readln (db_map,file_num,file_site,tuples_num);
                 {read first line of file description }
 writeln (file_num,file_site,tuples_num);
                 {read first line of file description }
 file_mtx[i].active := true ; { these file is active}
end;  {of with}

read (db_map,test_char);
new(x);  { dummy record to make the following loop work}
heads[i] := x ;  { heads(i) points to the first attribute
                              of file i in the log file }
last_att := x;
while test_char <> '*' do  { do while not end of
                                    attribute list}
begin
 {  creat an entry in the log file for a new file }
 new (x) ;
 last_att^.next_att := x ;  { connect previous record to
                                   the new record }
 last_att := x;    { points to the last attribute }
 last_att^.next_att := nil ;  { last attribute has
                                    no successor }
 last_att^.down := nil;

   with last_att^ do
   begin
   readln (db_map,att_num,att_name,att_type,key_att,
      value_range,actual_values,att_size); {read 2nd line}

   writeln (att_num,att_name,att_type,key_att,value_range,
                 actual_values,att_size); {read 2nd line }
   key_list[att_num] := key_att;   { prepare a list with
                             all of the key attributes }

   end ; { of with last_att^ }
{  fill in the file_mtx array }
   file_mtx[i].att_list[last_att^.att_num] := 1;
 new (y) ;       { creat an entry in the log file for a
                  specific value in the above attribute. }
 last_att^.down := y ; {points to the last value }
 last_value := y;
 current_value := y;
 read (db_map,test_char);
 while test_char <> '/' do  { while not end of value list}
 begin
```

```
   if last_att^.att_type = 1 then
      begin
       readln (db_map,current_value^.avalue,
                              current_value^.occurrences );
       writeln (current_value^.avalue,
                              current_value^.occurrences );
      end; {of if last_att^.att_type}
    if last_att^.att_type = 0 then
       begin
         readln (db_map,current_value^.nvalue,
                              current_value^.occurrences);
       end;   { of if last_att^.    }
   new (y);
   last_value := current_value;
   last_value^.down := y  ;
                { connect last value to the next one }
   current_value := y;
   current_value^.down := nil ;
                       { last value has no successor }

  read (db_map,test_char);
  end ;   { of while not '/' }
  last_value^.down := nil;  { eliminate an extra item
                                          at the end }

  readln(db_map);
  read (db_map,test_char);
 end ;  { of while not '*' }

readln(db_map);
read (db_map,test_char);
end ;   { of while not '$' }
file_mtx[i+1].file_num := 9999
end;  { of procedure read_data_file}
{**********************************************************}

procedure print_db;
{ this procedure prints a description of the data base.
The files that are printed are those left after the various
operations like Project, Select and Join.
}
var i :integer;
var  last_att : attribpointer ;
var  next_value : value_pointer ;

begin

 i := 0;       { counter for the file_mtx array }
```

```
while file_mtx[i+1].file_num <> 9999 do  { continue as
                             long as not end of files }
begin

i := i+1;    {increment counter by 1}
if file_mtx[i].active = true then
begin
{ each value of i represents another file in the
                          data base ( or empty file)}
last_att := heads[i]^.next_att;  { get the head of this
                                          structure }
with file_mtx[i] do
begin
writeln(db_out);
writeln(db_out,'              *       *       *       *',
                                    '       *       *');
writeln(db_out);
{  print the file's information contained in the
        file_mtx: file_num,file_site and tuples_num }
writeln (db_out,'    file # ',file_num:2,
                    '        site # ',file_site:2,
                '     has ',tuples_num:3:1, 'tuples ',
                '   file size is ',file_size : 5:2,
    ' parent 1 is ',parent1:2,' parent 2 is ',parent2:2);
writeln (db_out);
end;  {of with}

while last_att <> nil do { do while not end of
                                      attribute list}
begin

with last_att^ do
 begin
 writeln(db_out);
 writeln (db_out,'attribute#',att_num:2,' key= ',key_att,
'   name of att:',att_name,'   has',value_range:2,
'possible values, ','attr. size is ',att_size:2,' bytes');
   writeln(db_out);
 end ; { of with last_att^ }

   writeln (db_out,'    value        # of occurrences ');
   next_value := last_att^.down ;  { this is where the
                                        first value is }
  while next_value <> nil do   {do for all the values }
   begin

  if last_att^.att_type = 1 then   { alphanumerical value}
      writeln (db_out,'     ',next_value^.avalue,
                 '         ',next_value^.occurrences:4:2);
```

```
    if last_att^.att_type = 0 then  { numerical value }
     writeln (db_out,'      ',next_value^.nvalue:4,
                  '          ',next_value^.occurrences:4:2);

   next_value :=next_value^.down;
  end ;    { of while next_value <> nil }

  last_att := last_att^.next_att;
 end ;  { of while last_att <> nil }
end;     { of if file_mtx.active = 1}
end;     { of while heads[i+1] <> nil }

end;  { of procedure print_db }
{*************************************************************}
procedure read_cost(var cost_of_join,cost_of_tx :text);
{ this procedure reads the cost of sending a file from one
site another and the cost of join processing in each site }

var i,j,max_site : integer;

begin
max_site := 5;
reset (cost_of_join);
reset(cost_of_tx);
for i := 1 to max_site do
begin
read (cost_of_join,join_cost[i]);
write(join_cost[i] : 3);
end;
writeln;
for i := 1 to max_site do
begin
for j := 1 to max_site do
begin
 read (cost_of_tx,tx_cost[i,j]);
 write(tx_cost[i,j] : 3);
end;
readln(cost_of_tx);
writeln;
end;
end;   { of procedure }
{*************************************************************}
procedure print_file_mtx;

{ this procedure prints the file mtx }
var i,j : integer;

begin
i :=1;
```

```
while file_mtx[i].file_num <> 9999 do
begin

write (file_mtx[i].file_num :2 ,file_mtx[i].file_site :2 ,
                    file_mtx[i].tuples_num :6:2,'       ');
for j := 1 to 20 do
write (file_mtx[i].att_list[j] :2 );
writeln;

i := i+1
end   { of while file_mtx }

end;  { of procedure }



{*********************************************************}

procedure read_qry (var qry_map : text );
{ this procedure reads a description of a query from a text
file qry_map. The structure of this file is explained
earlier.
}

var i : integer ;
var end_of_qry : boolean;
begin
reset ( qry_map);{move pointer to the begining of the file}
i := 0;  { counter for the qry }
end_of_qry := false;
while end_of_qry = false do
begin
i := i+1;
{ fill in 1 line in the qry_mtx }
read( qry_map,qry_mtx[i].att_num);
if qry_mtx[i].att_num <> 9999 then
begin
read(qry_map,qry_mtx[i].att_type,
                      qry_mtx[i].must,qry_mtx[i].operator);
write( qry_mtx[i].att_num,qry_mtx[i].att_type,
                      qry_mtx[i].must,qry_mtx[i].operator);
if (qry_mtx[i].operator <> all ) then
begin           { this attribute should be selected }
 if (qry_mtx[i].att_type = 0 ) then   { numeric value }
 begin
   readln(qry_map,qry_mtx[i].nvalue)
 end;
 if (qry_mtx[i].att_type = 1 ) then  { alphanumeric value }
 begin
```

```
   readln(qry_map,qry_mtx[i].avalue);
   end
end;  { of if operator <> all }
writeln;
if (qry_mtx[i].operator = all) then
begin
    readln(qry_map)
end   {of if operator = all}
end     { of if att_num <> 9999}
else
end_of_qry := true
end    {of while end_of_qry}

end ; { of procedure read_qry }

{*********************************************************}

procedure calculate_size (filenum:integer);

{ this procedure gets as an input a file number. It scans
the linked list and calculates the size of the required
file. A file size is the sum of the atributes size in each
tuple times the number of the tuples in that file
}

var last_att,current_att:attribpointer;
var file_size ,tuple_size: real ;

begin
tuple_size := 0.0;
last_att := heads[filenum]; {get the header }
current_att := last_att^.next_att;  { get the first
                                     attribute in this file }

while  current_att <> nil do { do until end
                                          of attribute list }
begin
{  add the size of the attribute }
tuple_size :=  tuple_size + current_att^.att_size;
current_att := current_att^.next_att
end;  { of while }
{ multiply tuple_size by number of tuples in the file }
file_mtx[filenum].file_size :=
                  tuple_size * file_mtx[filenum].tuples_num

end;  { of procedure }


{*********************************************************}
```

```
procedure prepare_initial_projection ;

{ this procedure reads the structured array file_mtx and
  prepares 3 lists:
1)  all the attributes that appears in the query and are
    projected in the first step.( including the key
    attributes)
2)  all the attributes the must appear in the final answer.
3)  all the attributes that involes selection.
}
var i : integer;   { counter for qry_mtx array }
var j : integer;   { counter for Select_Mtx array }
begin
for i := 1 to 20 do
begin
initial_project_list [i] := 0;    { initialize the list }
project_after_select [i] := 0;
project_final[i] := 0;
end;  { of for i := 1 to 20 }
i := 0;
j := 0;
while qry_mtx[i+1].att_num <>9999 do
begin
i := i+1;
{set initial_project_list array according to the query }
initial_project_list[qry_mtx[i].att_num] := 1;

{prepare the list of attribures to be projected after the
selection phase and a list to be projected at the final
phase. right now these lists are identical ,at the end of
the procedure we add the key attributes to the
project_after_select list }
if (qry_mtx[i].must = 1) then
 begin
   project_after_select[ qry_mtx[i].att_num] := 1;
   project_final[ qry_mtx[i].att_num] := 1;
 end;

{prepare the matrix that contains the attributes that
                                        should be selected}

if qry_mtx[i].operator <> all then
 begin
  j := j+1;
  select_mtx[j].att_num  := qry_mtx[i].att_num;
  select_mtx[j].att_type := qry_mtx[i].att_type;
  select_mtx[j].must     := qry_mtx[i].must;
  select_mtx[j].operator := qry_mtx[i].operator;
```

```
  select_mtx[j].avalue   := qry_mtx[i].avalue;
  select_mtx[j].nvalue   := qry_mtx[i].nvalue
 end;      { of if qry_mtx.operator <> all }
select_mtx[j+1].att_num := 9999;  {terminate the list}

end;    {of while qry_mtx[i+1] }

{ add the key attributes to the list of attributes that
should be projected at the first step  and after the
selection}
for i := 1 to 20 do
begin
 if key_list[i] = 1 then
   begin
     initial_project_list[i] := 1;
     project_after_select[i] := 1;
   end;  { of if key_list[i] }
end;  { of for i := 1 to 20}

end;    { of procedure }


{**********************************************************}
procedure  check_if_need_to_project (proj_list : intlist;
                 i : integer; var proj_yes_no : boolean );

{ this procedure checks if a file need to be projected .}

var j : integer;

begin
proj_yes_no := false;

for j:= 1 to 20 do
begin
{ if need to project then proj_yes_no = true }
proj_yes_no := proj_yes_no or ((file_mtx[i].att_list[j]=1)
                                and ( proj_list[j] = 0));
end;
end;  {of procedure check_if_need}

{**********************************************************}

procedure disconnect_att(old_file:integer;new_file:integer;
                                     delete_list:intlist);
{ this procedure disconnects attribute which is not
required by the project operation}
var last_att,current_att:attribpointer;
```

```
begin
last_att := heads[old_file];
current_att := last_att^.next_att;
while  current_att <> nil  do
begin

{ if this attribute is in the list of attributes
                                        to be deleted }
if delete_list[current_att^.att_num] = 1 then
begin
{ delete (disconnect)this attribute from the linked list }
last_att^.next_att := current_att^.next_att;
dispose(current_att);    { return this node to the
                                        available list }
current_att := last_att^.next_att
end    { of if}

else begin  { of if delete_list ....    }
last_att := current_att;
current_att := current_att^.next_att
end     { of else}

end;  { of while }

heads[new_file] := heads[old_file];
heads[old_file] := nil;

end;  { of procedure }


{*******************************************************}
procedure delete_att (proj_list :intlist;old_file:integer);
{ this procedure creates a new line in the file_mtx,this
line contains the attributes after projection. The
procedure calls to procedures that update the linked list
and calculate the size of the new file
}
var j : integer;
var new_file : integer;

begin
new_file := last_file+1; { the value of last_value is set
                          at the procedure read_data_file }
last_file := new_file;

{ create a new line in the file_mtx array for
                                    the projected file }
file_mtx[new_file].file_num := new_file ;
file_mtx[new_file].file_site:=file_mtx[old_file].file_site;
```

```
file_mtx[old_file].active := false;
file_mtx[new_file].active := true;
file_mtx[new_file].tuples_num :=
                              file_mtx[old_file].tuples_num;
file_mtx[new_file].file_size :=
                               file_mtx[old_file].file_size;
file_mtx[new_file].parent1 := old_file;
file_mtx[new_file].parent2 := 0;

file_mtx[new_file+1].file_num := 9999;  {terminate the
                                                   file_mtx}


for j := 1 to 20 do
begin
{ create the new att_list after a file is projected }
file_mtx[new_file].att_list[j] :=
             file_mtx[old_file].att_list[j] * proj_list[j];

{ create a list that contains the numbers of the attributes
that should be deleted from the linked list }
if ((file_mtx[old_file].att_list[j] = 1)
                               and (proj_list[j] = 0)) then
delete_list[j] := 1


end;    { of for j:=1 to 20 }

{ call the procedure that scans the linked list and deletes
                                  the unrequired attribures}
disconnect_att ( old_file,new_file,delete_list);
calculate_size (new_file );
end;      { of procedure }

{***********************************************************}

procedure project (var proj_list : intlist);
{ this procedure gets as an input a list of attributes that
should be projected out of the files in the DB. The
procedure checks the files to make sure that projection is
needed,if it is so the procedure Delete_att is called. }

var proj_yes_no  : boolean;
i : integer;
old_file,new_file : integer;
begin
write(db_out,'  Project out attributes ');
for i := 1 to 20 do
  begin
```

```
    if proj_list[i] = 1 then
            write(db_out, i:3);
  end;
writeln(db_out);

i := 1;    { counter for the file_mtx[i] }

while file_mtx[i].file_num <> 9999 do
                              { as long as there are files }
begin
if file_mtx[i].active = true then
                         { check if this is an active file }
begin
 { check if file i need to be projected }
 check_if_need_to_project (proj_list,i,proj_yes_no);

 if proj_yes_no = true then  { yes, project upon this file}
  begin
  { for this file project the required attributes,create a
    new line in the file_mtx
  }
    old_file := i;

    delete_att ( proj_list,old_file);

  end;   { of if proj_yes_no}
end;     { of if file_mtx[i].active = true }
i := i+1;
end;  { of while }

end; { of procedure}

{***********************************************************}
procedure check_if_need_to_select (filenum,
       att_to_select:integer; var select_yes_no : boolean);
{ this procedure check if a file contains the required
attribute}

begin
select_yes_no := false; { initial value }
if ((file_mtx[filenum].active = true) and
     (file_mtx[filenum].att_list[att_to_select] = 1 )) then
select_yes_no := true ;
end;     { of procedure }

{+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
procedure skip_value (var last_value,current_value:
                                            value_pointer);
begin
```

```
   last_value:= current_value;
   current_value := current_value^.down;
 end ; { of function }
{+++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
procedure disconnect_value (var current_att:attribpointer;
               var last_value,current_value:value_pointer);
begin
        { if this is the first value then we need to change
                         the down field in the attribpointer }
 if current_att^.att_type = 1 then  { alpha numeric value }
begin
if (current_att^.down^.avalue = current_value^.avalue )then
             current_att^.down := current_att^.down^.down
          else   { 1 }
             last_value^.down := current_value^.down;
           current_value := current_value^.down;

    end  { of if current_att^.att_type }
  else   { att_type = 0  e.g. numeric value   }
    begin
if (current_att^.down^.nvalue = current_value^.nvalue )then
            current_att^.down := current_att^.down^.down
          else
           last_value^.down := current_value^.down;
       current_value := current_value^.down;
    end;  { of else of if current_att^.att_type }

end;    { of procedure}
{+++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
procedure count_tuple_num(filenum,att_to_select:integer;
                                  var adjust_factor:real);
{ The procedure counts the number of the remaining tuples
in the file after a selection or join is performed. The
procedure calculate the scaling factor which is used later
on to update the number of occurances of other attributes
in this file
}
var last_att,current_att : attribpointer;
var last_value,current_value : value_pointer ;
var num_of_tuples : real ;

begin
num_of_tuples := 0.0;
last_att := heads[filenum] ;
current_att := last_att^.next_att;
while current_att^.att_num <> att_to_select do
{  scan the linked list and find the required attribute }
begin
```

```
last_att := current_att;
current_att := current_att^.next_att;
end;  { of while }
last_value := current_att^.down;
current_value := last_value;

while current_value <> nil do
begin   { count the tuples associated with this attribute }
num_of_tuples :=num_of_tuples + current_value^.occurrences;
last_value:= current_value;
current_value := current_value^.down;
end;   { of while current_value}

adjust_factor :=num_of_tuples/file_mtx[filenum].tuples_num;
file_mtx[filenum].tuples_num := num_of_tuples ;
end;  { of procedure }

{*****************************************************}
procedure update_value_occurrences(filenum,att_to_select:
                              integer; adjust_factor:real);
{ The procedure is given an adjusting factor after a
selection was performed This factor is used to estimate the
number of occurrences of the rest of the values which
belong to other attributes
}
var last_att,current_att : attribpointer;
var last_value,current_value : value_pointer ;

begin
last_att := heads[filenum] ;
current_att := last_att^.next_att;
{ for all the attributes in this file do }
while current_att<> nil do
begin
{ skip the attribute on which a selection was performed }
if (current_att^.att_num <> att_to_select) then
begin
 current_value := current_att^.down;

 while current_value <> nil do
  begin { count the tuples associated with this attribute }
   current_value^.occurrences :=
            current_value^.occurrences * adjust_factor;
   current_value := current_value^.down;
 end;   { of while current_value}

end;  { of if  current_att.att_num <>att_to_saelect }
last_att := current_att;
current_att := current_att^.next_att;
```

```
end ;  { of while current_att <> nil }

end;   { of procedure }

{*****************************************************************}
```

```
procedure new_line (old_file:integer);
{ this procedure creates a new line in the file_mtx,this
line contains the attributes after selection.
}
var j : integer;
var new_file : integer;

begin
new_file := last_file+1; { the value of last_value is set
                        at the procedure read_data_file }
last_file := new_file;

{ create a new line in the file_mtx array for the
                                         projected file }
file_mtx[new_file].file_num := new_file ;
file_mtx[new_file].file_site :=
                            file_mtx[old_file].file_site;
file_mtx[old_file].active := false;
file_mtx[new_file].active := true;
file_mtx[new_file].tuples_num :=
                            file_mtx[old_file].tuples_num;
file_mtx[new_file].file_size :=
                              file_mtx[old_file].file_size;
file_mtx[new_file].parent1 := old_file;
file_mtx[new_file].parent2 := 0;

file_mtx[new_file+1].file_num := 9999; {end of file_mtx}

for j := 1 to 20 do
begin
{ create the new att_list after a file is projected }
file_mtx[new_file].att_list[j] := file_mtx[old_file].att_list[j];
end;

heads[new_file] := heads[old_file];
heads[old_file] := nil;

end; {of procedure }


{*********************************************************}
procedure select_file_upon_att(filenum,
      att_to_select: integer; select_mtx_line  : qryline );
{ this procedure scans the linked list,finds the required
attribute and eliminates the values which are not required
by the query.
}
var last_att,current_att : attribpointer;
var last_value,current_value : value_pointer ;
```

```
begin
last_att := heads[filenum] ;
current_att := last_att^.next_att;
while current_att^.att_num <> att_to_select do
{  scan the linked list and find the required attribute }
begin
last_att := current_att;
current_att := current_att^.next_att;
end;  { of while }
last_value := current_att^.down;
current_value := last_value;

while current_value <> nil do
begin
case select_mtx_line.operator of

eq :                  {        select_upon_eq;    }
   begin
   if current_value^.avalue = select_mtx_line.avalue then
    skip_value(last_value,current_value)
   else
    disconnect_value(current_att,last_value,current_value);
  end;  { of eq:}
gt :                  {        select_upon_eq;    }
   begin
   if current_value^.nvalue >= select_mtx_line.nvalue then
    skip_value(last_value,current_value)
   else
    disconnect_value(current_att,last_value,current_value);
  end;  { of gt:}

end;  {of case}

end;   { of while}

end; {of procedure }

{**********************************************************}

procedure select_all_files ;

{ this procedure scans the files in the DB and selects
attribures according to a pre defined list of attributes
and required values .
}
var jj ,filenum,att_to_select : integer;
var select_yes_no : boolean;
select_factor : real;
```

```
begin
jj := 1 ;   {counter for the lines in the select matrix}

while select_mtx[jj].att_num <> 9999 do { for each line in
                                        the select_mtx do}
begin
att_to_select := select_mtx[jj].att_num;
for filenum := 1 to last_file do { work on each file
                                              in the DB}
 begin
{ check if the file contains the attribute we want
                                          to select upon }
  check_if_need_to_select (filenum,att_to_select,
                                           select_yes_no);
  if select_yes_no = true  then { this file has the
                                      required attribute }

   begin
    { perform the selection }
    select_file_upon_att (filenum,att_to_select,
                                          select_mtx[jj]);
    new_line(filenum);
    { calculate how many tuples are in the file after
                                              selection }
    count_tuple_num(last_file,att_to_select,select_factor);

    { adjust the number of occurrences of the other values
                                           in this file }
    update_value_occurrences(last_file,att_to_select,
                                           select_factor);
    { calculate the size of the file after selection }
    calculate_size(last_file);
   writeln(db_out,'Select file',filenum:2,'upon attribute',
             att_to_select:2,' to get file ',last_file:2);
   end;  { of if_select_yes_no = true }
 end;    { of for filenum := 1 to last_file }
 jj := jj+1;

end;       { of while  }
end;        { of procedure }
{*********************************************************}
procedure new_line_after_join(old_file1,old_file2:integer);
{ this procedure creates a new line in the file_mtx,this
line contains the attributes of two joined files.
}
var j : integer;
var new_file : integer;

begin
```

```
new_file := last_file+1; { the value of last_value is set
                           at the procedure read_data_file }
last_file := new_file;

{ create a new line in the file_mtx array for
                                    the projected file }
file_mtx[new_file].file_num := new_file ;
file_mtx[new_file].file_site :=
                             file_mtx[old_file2].file_site;
file_mtx[new_file].tuples_num :=
                            file_mtx[old_file1].tuples_num;
file_mtx[old_file1].active := false;
file_mtx[old_file2].active := false;
file_mtx[new_file].active := true;
file_mtx[new_file].parent1 := old_file1;
file_mtx[new_file].parent2 := old_file2;

file_mtx[new_file+1].file_num := 9999;
                                    {terminate the file_mtx}

for j := 1 to 20 do
begin
{ create the new att_list after a file is joined }

if ( file_mtx[old_file1].att_list[j] = 1 ) or (
                 file_mtx[old_file2].att_list[j] = 1 ) then
file_mtx[new_file].att_list[j] := 1;
end;

heads[new_file] := heads[old_file1];
heads[old_file1] := nil;
heads[old_file2] := nil;

writeln(db_out);
writeln(db_out, '  Join files ',old_file1:2,' and ',
    old_file2:2, ' to get the new file ',new_file:2);
writeln(db_out,'  The price of this join is ',
                                    ip_mtx[0,2*kmax+1] :8);
end; {of procedure }

{*******************************************************}
procedure check_if_common_att(file1,file2 : integer;
                                  var common_att:integer);
{ this procedure scans the file_mtx and looks for a common
attribute within two files.
}
var i : integer;
begin
common_att := 0;   { initial value }
```

```
for i := 1 to 20 do
begin
if (( file_mtx[file1].att_list[i] = 1 ) and
                  ( file_mtx[file2].att_list[i] = 1 )) then
common_att := i;
end;   { of for i }

end;   { of procedure }


{*********************************************************}
procedure check_if_common_value(file1,file2,
          common_att:integer; var joinable_yes_no:boolean);
{ this procedures gets the names of two files and and
checks if these files have common values within a common
attribute. If there are such two values then these files
are joinable.
}
var last_att1,current_att1 : attribpointer;
var last_value1,current_value1 : value_pointer ;
var last_att2,current_att2 : attribpointer;
var last_value2,current_value2 : value_pointer ;

begin
{ scan the linked list and find the pointers to the common
attribute in both files ,file1 and file2 .
}

last_att1 := heads[file1] ;
current_att1 := last_att1^.next_att;
while current_att1^.att_num <> common_att do
{scan the linked list and find the required 1st. attribute}
begin
last_att1 := current_att1;
current_att1 := current_att1^.next_att;
end;  { of while }

last_att2 := heads[file2] ;
current_att2 := last_att2^.next_att;
while current_att2^.att_num <> common_att do
{scan the linked list and find the required 2nd. attribute}
begin
last_att2 := current_att2;
current_att2 := current_att2^.next_att;
end;  { of while }

joinable_yes_no := false;  { initial value for this value }

{ for the common_attribute of file1,scan the values of this
```

```
attribute. Compare each value with the values of common_att
of file2. If there is at match at least once we say that
these two files are joinable.
}
last_value1 := current_att1^.down;
current_value1 := last_value1;

while current_value1 <> nil do
begin
last_value2 := current_att2^.down;
current_value2 := last_value2;

while current_value2 <> nil do
 begin
  if current_att1^.att_type = 1 then{if alphanumeric value}
   begin
  if (current_value1^.avalue = current_value2^.avalue) then
     joinable_yes_no := true;
   end
    else    { if numeric value }
   begin
  if (current_value1^.nvalue = current_value2^.nvalue) then
       joinable_yes_no := true;
   end;
    skip_value(last_value2,current_value2);
  end;   { of while current_value2 <> nil }
  skip_value(last_value1,current_value1);
                              { go to next value of attrib 1}
end;     { of while current_value1 <> nil }

end;        { of procedure }

{*************************************************************}
procedure check_if_joinable(f1,f2 :integer;
      var common_att : integer; var is_joinable : boolean);
{ this procedure calls procedures to check if the two files
has a common attribute and common values }

begin
is_joinable := false;
check_if_common_att(f1,f2,common_att);
if common_att <> 0 then
begin
check_if_common_value(f1,f2,common_att,is_joinable);

end; { if common_att }
end;   { of procedure }
{*************************************************************}
procedure create_join_mtx;
```

```
{ for every pair of active files in the DB this procedure
will check the joinability of the two files. If joinable,
the procedure will fill in a column in the join_mtx . This
column contains the file # , the file site and the common
attribute.
}
var f1,f2,column,common_att : integer;
var is_joinable : boolean;
var i,j : integer;

begin

{ erase the join matrix }
for i := 1 to 5 do
begin
for j := 1 to 20 do
  join_mtx[i,j] := 0;
end;
f1 := 1;   column := 0;
while file_mtx[f1].file_num <> 9999 do
begin
f2 := f1+1;
while file_mtx[f2].file_num <> 9999 do
 begin
  if ((file_mtx[f1].active = true ) and
                        (file_mtx[f2].active =true))  then
   begin
     check_if_joinable(f1,f2,common_att,is_joinable );
     if is_joinable = true then
      begin
       column := column + 1;
                    { index for the columns of the join_mtx}
       join_mtx[1,column] := file_mtx[f1].file_site;
       join_mtx[2,column] := file_mtx[f2].file_site;
       join_mtx[3,column] := f1;
       join_mtx[4,column] := f2;
       join_mtx[5,column] := common_att;

{ if f1 is joinable with f2 then f2 is joinable with f1 }
        column := column + 1; { index for the columns
                                            of the join_mtx}
        join_mtx[1,column] := file_mtx[f2].file_site;
        join_mtx[2,column] := file_mtx[f1].file_site;
        join_mtx[3,column] := f2;
        join_mtx[4,column] := f1;
        join_mtx[5,column] := common_att;
    end;  { of if joinable_yes_no = true }
   end; { of if file_mtx[f1].active=true and
                                  file_mtx[f2].active =true}
```

```
  f2 := f2 + 1;
end;    { of while file_mtx[f2].file_num <> 9999 }

f1 := f1 + 1;
end;   { of while file_mtx[f1].file_num <> 9999 }


end_join := false;
if (join_mtx[1,1] = 0 )then
    end_join := true;
end;   { of procedure }
{*******************************************************************}
procedure join_two_files(file1,file2,common_att:integer);
{ this procedures gets the names of two files and joins
them accoroding to a common attribute.
}
var last_att1,current_att1 : attribpointer;
var last_value1,current_value1 : value_pointer ;
var last_att2,current_att2 : attribpointer;
var last_value2,current_value2 : value_pointer ;
var match : boolean;
var join_factor : real;

begin
{ scan the linked list and find the pointers to the common
attribute in both files ,file1 and file2
}
last_att1 := heads[file1] ;
current_att1 := last_att1^.next_att;
while current_att1^.att_num <> common_att do
{scan the linked list and find the required 1st. attribute}
begin
last_att1 := current_att1;
current_att1 := current_att1^.next_att;
end;  { of while }

last_att2 := heads[file2] ;
current_att2 := last_att2^.next_att;
while current_att2^.att_num <> common_att do
{scan the linked list and find the required 2nd. attribute}
begin
last_att2 := current_att2;
current_att2 := current_att2^.next_att;
end;  { of while }

{ for the common_attribute of file1,scan the values of this
attribute. Compare each value with the values of common_att
of file2. If there is a match, we update the number of
```

```
occurrences of this value to be the product of the number
of occurrences of each value. If there is no match ,this
value is eliminated from the linked list.
}
last_value1 := current_att1^.down;
current_value1 := last_value1;

if current_att1^.key_att =0 then    { do only for non
                                          key attribute}
begin
while current_value1 <> nil do
begin
match := false;
last_value2 := current_att2^.down;
current_value2 := last_value2;

while (current_value2 <> nil) and (match = false ) do
 begin
  if current_att1^.att_type =1 then {if alphanumeric value}
   begin
   writeln('FROM JOIN TWO value1  ',
current_value1^.avalue,' value2  ',current_value2^.avalue);
  if (current_value1^.avalue = current_value2^.avalue) then
    begin
    match := true;
    current_value1^.occurrences :=
 current_value1^.occurrences * current_value2^.occurrences;
    end;  { of if current_value^.avalue =        }
     skip_value(last_value2,current_value2);

   end
    else    { if numeric value }
     begin
  if (current_value1^.nvalue = current_value2^.nvalue) then
       begin
        match := true;
        current_value1^.occurrences :=
 current_value1^.occurrences * current_value2^.occurrences;
       end;
     skip_value(last_value2,current_value2);

     end;   { of else }
  end;   { of while current_value2 <> nil }

  if match = true then
   skip_value(last_value1,current_value1)
                              { go to next value of attrib 1}
    else
 disconnect_value(current_att1,last_value1,current_value1);
```

```
end;     { of while current_value1 <> nil }
end    { of if cuurent_att.key_att = 0}
else    { e.g this is a key attribute }
{ assume that key attributes are uniformly distributed then
the new occurrences number is given by: occurrences1 *
occurrences2/ value_range }

        last_value2 := current_att2^.down;
        current_value2 := last_value2;
                                { point to the 2nd key value}
        current_value1^.occurrences :=
     current_value1^.occurrences *
     current_value2^.occurrences/current_att1^.value_range;

    { calculate how many tuples are in f1 after join }

    count_tuple_num(file1,common_att,join_factor);
    { adjust the number of occurrences of the other
                                        values in this file }
    update_value_occurrences(file1,common_att,join_factor);

{ f2 contains the same number of tuples as f1 , we can
calculate the join factor for f2 as follow }

 join_factor :=
   file_mtx[file1].tuples_num / file_mtx[file2].tuples_num;

   file_mtx[file2].tuples_num := file_mtx[file1].tuples_num;

    { adjust the number of occurrences of the other values
                                               in this file }
    update_value_occurrences(file2,common_att,join_factor);

{ after the common attribute is worked on we need to
connect the two files and eliminate the extra column (
remember that the two files have the same attribute)
}

last_att2^.next_att := current_att2^.next_att;
                                   { eliminate an attribute }

{ scan the linked list and find the end of the 1st file }

while current_att1^.next_att <> nil do
begin
last_att1 := current_att1;
current_att1 := current_att1^.next_att;
end;  { of while current }
```

```
{ connect end of file 1 to the beginning of file 2 }

current_att1^.next_att := heads[file2]^.next_att;
heads[file2] := nil;

new_line_after_join(file1,file2);


{ calculate the size of the joined file }
calculate_size(last_file);

end;          { of procedure }

{********************************************************}


procedure create_lp_mtx( xi: integer ;
                              var nnn,max_var : integer );
{ this procedure creates the lp matrix.
The first line in this matrix determines the lp function.
The rest of the lines in the matrix determine the
constraints on this function.
}
var k,ff,lp_counter : integer;
var pos : boolean;

            {  create the lp function  }
            {--------------------------}
begin
k:= 1;          { set the X's in the function }
while join_mtx[1,k] <> 0 do
begin
{ ip_mtx[0,k] := tx_cost[m] *  size(i)  }
ip_mtx[0,k] := round(tx_cost[join_mtx[1,k],join_mtx[2,k]] *
                      file_mtx[join_mtx[3,k]].file_size );
k := k+1;
end;   { of while }
kmax := k-1;   { how many columns in join_mtx}



k:= 1;          { set the Z's in the function }
while join_mtx[1,k] <> 0 do
begin
{ ip_mtx[0,k+kmax] := join_cost[n] * ( size(i) + size(j) )}
ip_mtx[0,k+kmax] := round ( join_cost[join_mtx[2,k]] *
(file_mtx[join_mtx[3,k]].file_size +
                      file_mtx[join_mtx[4,k]].file_size) );
```

```
k := k+1;
end;    { of while }


              {  create the 1st constraint }
              {----------------------------}
{  Z(m,n,i,j) + Z(n,m,j,i) <= Y(m,n,i,j)   }
lp_counter := 1;     { counter points to the 2nd line in the
                                                    lp matrix}

k:= 1;
while join_mtx[1,k] <> 0 do
begin

ip_mtx[lp_counter,k+kmax] := 1;

{ for each set of m,n,i,j  find in which column is
                                        the set n,m,j,i }
ip_mtx[lp_counter,k+1 + kmax] := 1;
ip_mtx[lp_counter,2*kmax+1] := 1;      { set the Y's }
k := k+2;
lp_counter := lp_counter + 1;
end;  { of while }


              {  create the  constraint }
              {----------------------------}
{  X(m,n,i,j) - Z(n,m,j,i) = 0 }
{  we write an equality as a set of two inequalities   }


{  X(m,n,i,j) - Z(n,m,j,i) <= 0 }

pos:= true;
k:= 1;
while join_mtx[1,k] <> 0 do
 begin
if join_mtx[1,k] <> join_mtx[2,k] then  { only if m <> n }
begin
 ip_mtx[lp_counter,k] := 1;
 ip_mtx[lp_counter,k+kmax] := -1;
end; { of if join_mtx[ ] <>    }
k := k+1;
pos := not pos;
lp_counter := lp_counter + 1;
end;   { of while }


{ -X(m,n,i,j) + Z(n,m,j,i) <= 0 }
```

```
pos:= true;
k:= 1;
while join_mtx[1,k] <> 0 do
begin

ip_mtx[lp_counter,k] := -1;
ip_mtx[lp_counter,k+ kmax ] := 1;
k := k+1;
pos := not pos;
lp_counter := lp_counter + 1;
end;   { of while }


          { create a constraint }
      { sum of all Z(m,n,i,j) = xi }

k:= 1;
while join_mtx[1,k] <> 0 do
begin
ip_mtx[lp_counter,k+kmax] := 1;
k := k+1;

end;
ip_mtx[lp_counter,2*kmax+1] := xi;
lp_counter := lp_counter + 1;

{  multiply both sides by -1 }
k:= 1;
while join_mtx[1,k] <> 0 do
begin
ip_mtx[lp_counter,k+kmax] := -1;
k := k+1;

end;
ip_mtx[lp_counter,2*kmax+1] := -xi;
lp_counter := lp_counter + 1;


              { create a constaint }
          { a file can be joined with only one other file }
{ scan the join_mtx, for each active file  create a line
which contains all the Z's with either i or j are equal to
this number}

for ff := 1 to last_file do
begin
if file_mtx[ff].active = true then
begin
```

```
   for k := 1 to kmax do
    begin
     if ( join_mtx[3,k] = ff) or( join_mtx[4,k] = ff ) then
       begin
         ip_mtx[lp_counter,k + kmax ] := 1;

       end;  { of if join_mtx[3,k]     }
   end;  { for k := 1 to kmax }
   ip_mtx[lp_counter, 2*kmax + 1] := 1;
   lp_counter := lp_counter+1;
 end;    { of if file_mtx.active }
 end;     { of for ff := 1 to last file}

         {    create a constraint    }
     { X(m,n,i,j) = ( 0,1)     Z(m,n,i,j) = ( 0,1)   }

   for k := 1 to kmax do
    begin
     ip_mtx[ lp_counter,k] := 1;
     ip_mtx[ lp_counter,2*kmax+1] := 1;      { X < 1 }
     lp_counter := lp_counter+1;

     ip_mtx[ lp_counter,k+kmax] := 1;
     ip_mtx[ lp_counter,2*kmax+1] := 1;     { Z < 1}
     lp_counter := lp_counter+1;

     ip_mtx[ lp_counter,k] := -1;              { -X < 0 }
     lp_counter := lp_counter+1;

     ip_mtx[ lp_counter,k+kmax] := -1;              { -Z < 0 }
     lp_counter := lp_counter+1;
    end;                               { of for k := 1 to kmax }

     max_var := 2*kmax;     { number of varriables }
    nnn := lp_counter -1 + max_var; { a parameter for the
     allinteger procedure e.g number of variables + number
     of constraints}

{ after completion of the constraints we create the I
matrix at the end of the ip_mtx as required by the
allinteger procedure}

   for k := 1 to 2*kmax do
    begin
     ip_mtx[lp_counter,k] := -1 ;
     lp_counter := lp_counter + 1;
    end;   { of for k}
writeln(' xi  xi =',xi);
end;
```

```
{****p***************************************************}
procedure allinteger(
m,n            :integer;
var  a         :arrmn1;
var  count     :integer;
var  nofeas    :boolean);

{ this procedure solves the LP model as defined in the
IP_MTX array.}

var  c,denom,i,j,k,l,np,num,r,r1,s,t  :integer;
 b,iter                              :boolean;

function euclid(u,v :integer) : integer;
var  w :integer;
begin
w:= u div v;
if w*v > u then w:=w-1;
if (w+1)*v <= u then w:=w+1;
euclid :=w
end;   {end of euclid }

begin
{
reset (datain);
for i:= 0 to m do
begin
 for j:= 1 to n+1 do
   read (datain ,a[i,j])
end;
}
rewrite (result);
for i:= 0 to m do
begin
 writeln (result);
 for j:= 1 to n+1 do
   write (result ,a[i,j]:3)
end;


count:=0;
np:= n+1;
repeat
  count:=count+1;
  r:=0;
  repeat
     r:=r+1;
     iter:=a[r,np] < 0
  until iter or (r = m);
```

```
  if iter then
  begin
     k:=0;
     repeat
        k:= k+1;
        iter:= a[r,k] < 0
     until iter or (k = n);
     nofeas:= not iter;
     if iter then
     begin
        l:=k;
        for j:=k+1 to n do
            if a[r,j] < 0 then
            begin
               i:= -1;
               repeat
                  i := i+1;
                  s := a[i,j]-a[i,l]
               until s <> 0 ;
               if s < 0 then l:= j
            end;
s:= 0;
while a[s,l]=0 do s:=s+1;
num:=-a[r,l];
denom:=1;
for j:=1 to n do
    if (a[r,j] < 0 ) and (j<>l) then
    begin
       i:=s-1;
       b:=true;
       while b and (i>=0) do
       begin
          b:=a[i,j]= 0;
          i:=i-1
       end;
       if b then
       begin
          i:=a[s,j];
          r1:=a[s,l];
          t:=euclid(i,r1);
          if (t*r1 = i ) and (t>1) then
          begin
             i:=s;
             repeat
                i:=i+1;
                r1:=t*a[i,l]-a[i,j]
              until r1 <> 0;
              if r1 > 0 then t:=t-1
          end;
```

```
            c:=-a[r,j];
            if c*denom > t*num then
            begin
               num:=c;
               denom:=t
            end
         end              { b }
     end;                 { j }
    for j:=1 to np do
        if j <>1 then
        begin
           c:=euclid(a[r,j]*denom,num);
           if c<> 0 then
              for i:=0 to m do
                  a[i,j]:=a[i,j]+c*a[i,1]
        end
    end          { iter: a[r,k] < 0 }
 end             { iter: a[r,np] < 0 }
until not iter or nofeas;
writeln(result);
writeln(result,'count= ',count,
             '    nofeas= ',nofeas,' m= ',m,'  n= ',n );

writeln (result);
writeln (result);

writeln (result,'     -a[0,n+1]= ',-a[0,n+1] );
for j:= 1 to n  do
writeln ( result,a[m-n+j,n+1] ) ;
for i:= 0 to m do
begin
 writeln (result);
 for j:= 1 to n+1 do
   write (result ,a[i,j] :3);
end;
writeln (result,'    count= ',count,'  nofeas= ',nofeas )
end  ; { of allinteger }

{**********************************************************}
procedure print_lp_mtx;
var i,j : integer;
begin


writeln(lp,'        JOIN MATRIX ');
for i := 1 to 5 do
begin

for j := 1 to 20 do
```

```
begin
write(lp,join_mtx[i,j]:4);
end;
writeln(lp);
end;
writeln(lp);
writeln(lp);
writeln(lp,'     THE LP FUNCTION  ');
writeln(lp);
for j := 1 to 10 do
begin
 write (lp,ip_mtx[0,j] : 8 );
end;
writeln(lp);

for j := 11 to 22 do
begin
 write (lp,ip_mtx[0,j] : 8 );
end;
writeln(lp);

for i := 1 to 100 do
begin
for j := 1 to 25 do
begin
 write (lp,ip_mtx[i,j] : 3 );
end;
writeln(lp);
end;
end;
{**********************************************************}
procedure erase_lp_mtx;
var i,j : integer;

begin
for i := 0 to 100 do
begin
for j := 1 to 100 do
 ip_mtx[i,j] := 0;
end;
end;

{**********************************************************}
procedure determine_join;
{ this procedure examines the files in the data base and
with the help of the lp procedure determines which files
are to b joined}

var xi : integer;
```

```
begin
xi := 1; { number of joins we want to perform. this value
    is to "1" in this version but other values might be
                                              later examined}
erase_lp_mtx;
create_lp_mtx(xi,mm,nn);
print_lp_mtx;
allinteger(mm,nn,ip_mtx,count1,no_solution);
print_lp_mtx;

end;   { of determine }

{**********************************************************}
procedure activate_join;
{ this procedure is called after the cheapest join is
determine by the LP procedure. The results of the LP
procedure ( e.g the Zi that are assigined the value "1" by
the LP procedure ) are kept in the global array ip_mtx.
This procedure retreives the values of the Zi's and call a
procedure to perform the join if neccessary }

var i,j,f1,f2,comn_att : integer;

begin
{ the solution vector is placed by the LP procedure starting
at row mm-2*kmax + 1 to row mm, in column 2*kmax+1
}
for i := mm-kmax + 1   to mm do
begin
j := i- ( mm-kmax);
if ip_mtx[i,2*kmax+1] = 1 then
begin
  f1 := join_mtx[3,j];
  f2 := join_mtx[4,j];
  comn_att := join_mtx[5,j];
  join_two_files(f1,f2,comn_att);

end;  { of if ip_mtx }
end;    { of for }
end;   { of procedure }


{**********************************************************}

          {*******   MAIN PROGRAM STARTS HERE   ******}

begin
rewrite (db_out); {move pointer to the begining of the file
```

```
    db_out.dat this file contains the answer to the query }
read_data_file (db_map);  { read the structer of the
                                              data file }

read_cost(cost_of_join,cost_of_tx);  { read join
                                cost and transmission cost}
rewrite(lp);

for ii := 1 to last_file do
calculate_size(ii);    {calculate the size of each file }
print_db;
print_file_mtx;
read_qry (qry_map);
prepare_initial_projection;
print_file_mtx;
writeln(db_out);
writeln(db_out,'      DATA BASE AFTER INITIAL PROJECTION');
writeln(db_out);
project(initial_project_list);
print_db;

writeln(db_out);
writeln(db_out,'             DATA BASE AFTER SELECTION');
select_all_files;
print_db;
project(project_after_select);

writeln(db_out);
writeln(db_out,'          DATA BASE AFTER 2nd PROJECTION');
writeln(db_out);
print_db;

while not end_join do
begin
create_join_mtx;
if end_join = false then
begin
 determine_join;
 writeln(db_out);
 writeln(db_out,'              DATA BASE AFTER JOIN ');
 writeln(db_out);
 activate_join;
 print_db;
end { of if end_join}
end;  { of while }

writeln(db_out);
writeln(db_out,'          DATA BASE AFTER 3rd PROJECTION');
writeln(db_out);
```

```
project(project_final);    { remove key attributes}
print_db;
end.
```

A P P E N D I X C

C.1 An Example of the Implementation of the Heuristic Based Algorithm

The following example illustrates the various steps that are taken during the query optimization process.

We are given the following information: at site 1 we have relation 1; at site 2, relation 2; and at site 3, relations 3 and 4.

The relations are as follows:

f1: Supplier Relation at site 1

Attributes (sizes in bytes)	S# (2)	Sname (5)	Status (2)	Scity (6)
	S1	Smith	20	London
	S2	Jones	10	Paris
	S3	Blake	30	Paris
	S4	Clark	20	London
	S5	Adams	30	Athens

f2: Parts Relation at site 2

Attributes (sizes in bytes)	P# (2)	Pname (5)	Color (5)	Weight (2)	Pcity (6)
	P1	Nut	Red	12	London
	P2	Bolt	Green	17	Paris
	P3	Screw	Blue	17	Rome
	P4	Screw	Red	14	London
	P5	Cam	Blue	12	Paris
	P6	Cog	Red	19	London

f3: Projects Relation at site 3

Attributes (sizes in bytes)	J# (2)	Jname (8)	Jcity (6)
	J1	Sorter	Paris
	J2	Reader	Rome
	J3	Console	Athens
	J4	Collator	London
	J5	Terminal	Oslo
	J6	Tape	London

f4: Supplier-Parts-Project Relation at site 3

Attributes (sizes in bytes)	S# (2)	P# (2)	J# (2)	Qty (3)
	S1	P1	J1	200
	S1	P1	J4	700
	S2	P3	J1	400
	S2	P3	J2	200
	S2	P3	J3	200
	S2	P3	J4	500
	S2	P3	J5	600
	S2	P3	J6	400
	S2	P3	J7	800
	S2	P5	J2	100
	S3	P3	J1	200

Based upon the above information, the statistics on the relations are extracted and placed in the log file. The initial set up of the log file is described below.

INITIAL SET UP OF THE LOG FILE

* * * * * *

file #1 at site #1 has 5.0tuples, file size is 75.00 bytes.

attribute #1 is a key attribute, name of att.: s#, has 5 possible values, attribute size is 2 bytes.

value	# of occurrences
s#	5.00

attribute #2 - name of att: sname.
attribute size is 5 bytes.

value	# of occurrences
smith	1.00
jones	1.00
blake	1.00
clark	1.00
adams	1.00

attribute #3 - name of att: status.
attribute size is 2 bytes.

value	# of occurrences
20	2.00
10	1.00
30	2.00

attribute # 4 name of att: scity
attribute size is 6 bytes.

value	# of occurrences
london	2.00
paris	2.00
athens	1.00

* * * * * *

file #2 at site #2 has 6.0tuples, file size is 120.00.

attribute #5 is a key attribute, name of att: p#, has 6 possible values, attribute size is 2 bytes.

value	# of occurrences
p#	6.00

attribute #6 - name of att: pname,
attribute size is 5 bytes.

value	# of occurrences
nut	1.00
bolt	1.00
screw	2.00
cam	1.00
cog	1.00

attribute #7 name of att: color,
attribute size is 5 bytes.

value	# of occurrences
red	3.00
green	1.00
blue	2.00

attribute #8 - name of att: weight,
attribute size is 2 bytes

value	# of occurrences
12	2.00
14	1.00
17	2.00
19	1.00

attribute #9 - name of att: pcity,
attribute size is 6 bytes.

value	# of occurrences
london	3.00
paris	2.00
rome	1.00

* * * * * *

file #3 at site #3 has 6.0tuples, file size is 96.00

attribute # 10 is a key attribute, name of att: J#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	6.00

attribute # 11 - name of att: jname ,
attribute size is 8 bytes.

value	# of occurrences
sorter	1.00
reader	1.00
console	1.00
collator	1.00
terminal	1.00
tape	1.00

attribute # 12 - name of att: jcity, 6 bytes

value	# of occurrences
paris	1.00
rome	1.00
athens	1.00
london	2.00
oslo	1.00

* * * * * *

file #4 at site #3 has 11.0 tuples, file size is 99.00 bytes.

attribute #1 is a key attribute, name of att: s#, has 5 possible values, attribute size is 2 bytes.

value	# of occurrences
s#	11.00

attribute #5 is a key attribute, name of att: p#, has 6 possible values, attribute size is 2 bytes.

value	# of occurrences
p#	11.00

attribute #10 is a key attribute, name of att: j#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	11.00

attribute #13 - name of att: qty,
attribute size is 3 bytes.

value	# of occurrences
100	1.00
200	4.00
400	2.00
500	1.00
600	1.00

700	1.00
800	1.00

A query originates at site 1 requesting the attributes Sname, Scity, Weight, and Quantity and the predicate relationships: Pname = Screws, Quantity > 400, and Jcity = Athens.

Attributes 1, 2, 4, 5, 6, 8, 10, 12, 13 include all the attributes which are specified by the query along with key attributes. These attributes are projected out as a first step in the query processing.

LOG FILE AFTER INITIAL PROJECTION

* * * * * *

file #4 at site #3 has 11.0tuples, file size is 99.00 bytes.

attribute #1 is a key, name of att: s#, has 5 possible values, attribute size is 2 bytes.

value	# of occurrences
s#	11.00

attribute #5 is a key, name of att: p#, has 6 possible values, attribute size is 2 bytes.

value	# of occurrences
p#	11.00

attribute #10 is a key, name of att: j#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	11.00

attribute # 13 - name of att: qty, attribute size is 3 bytes.

value	# of occurrences
100	1.00
200	4.00
400	2.00
500	1.00
600	1.00

```
 700          1.00
 800          1.00

       *       *       *       *       *       *

file # 5 at site # 1 has 5.0tuples, file size is 65.00
parent 1 is file 1


attribute # 1 is a key, name of att: s#, has 5 possible
values, attribute size is 2 bytes.

value                # of occurrences
 s#                          5.00

attribute # 2 - name of att: sname, has 5 possible values,
attribute size is 5 bytes.

value                # of occurrences
 smith                       1.00
 jones                       1.00
 blake                       1.00
 clark                       1.00
 adams                       1.00

attribute # 4 - name of att: scity, has 5 possible values,
attribute size is 6 bytes.

value                # of occurrences
 london                      2.00
 paris                       2.00
 athens                      1.00

       *       *       *       *       *       *

file # 6 at site # 2 has 6.0tuples, file size is 54.00,
parent 1 is file 2


attribute # 5 is a key, name of att: p#, has 6 possible
values, attribute size is 2 bytes.

value                # of occurrences
 p#                          6.00

attribute # 6, name of att: pname,
attribute size is 5 bytes.

value                # of occurrences
```

nut	1.00
bolt	1.00
screw	2.00
cam	1.00
cog	1.00

attribute # 8, name of att: weight, has 30 possible values, attribute size is 2 bytes.

value	# of occurrences
12	2.00
14	1.00
17	2.00
19	1.00

*　　*　　*　　*　　*　　*

file # 7 at site # 3 has 6.0 tuples, file size is 48.00 bytes, parent 1 is file 3

attribute # 10 is a key, name of att: J#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	6.00

attribute # 12, name of att: jcity, has10 possible values, attribute size is 6 bytes.

value	# of occurrences
paris	1.00
rome	1.00
athens	1.00
london	2.00
oslo	1.00

After all of the projections are completed we perfom the selections required by the query. Thus we want only the tuples which satisfy the following restrictions :

Attribute #6 PNAME = SCREWS
Attribute #12 JCITY = ATHENS
Attribute #13 QTY > 400

LOG FILE AFTER SELECTION

* * * * * *

file # 5 at site # 1 has 5.0 tuples, file size is 65.00,
parent 1 is file 1.

attribute # 1 is a key, name of att: s#, has 5 possible values, attribute size is 2 bytes.

value	# of occurrences
s#	5.00

attribute # 2, name of att: sname,
attribute size is 5 bytes.

value	# of occurrences
smith	1.00
jones	1.00
blake	1.00
clark	1.00
adams	1.00

attribute # 4, name of att: scity,
attribute size is 6 bytes.

value	# of occurrences
london	2.00
paris	2.00
athens	1.00

* * * * * *

file # 8 at site # 2 has 2.0 tuples, file size is 18.00
parent 1 is file 6

attribute # 5 is a key, name of att: p#, has 6 possible values, attribute size is 2 bytes.

value	# of occurrences
p#	2.00

attribute # 6, name of att: pname, has 5 possible values,
attribute size is 5 bytes.

value	# of occurrences
screw	2.00

attribute # 8, name of att: weight, 2 bytes.

value	# of occurrences
12	0.67
14	0.33
17	0.67
19	0.33

* * * * * *

file # 9 at site # 3, has 6.0 tuples, file size is 54.00
parent 1 is file 4.

attribute # 1 is a key, name of att: s#, has 5 possible values, attribute size is 2 bytes.

value	# of occurrences
s#	6.00

attribute # 5 is a key, name of att: p#, has 6 possible values, attribute size is 2 bytes.

value	# of occurrences
p#	6.00

attribute # 10 is a key, name of att: j#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	6.00

attribute # 13, name of att: qty, has 10 possible values, attribute size is 3 bytes.

value	# of occurrences
400	2.00
500	1.00
600	1.00
700	1.00
800	1.00

* * * * * *

file # 10 at site # 3 has 1.0 tuple, file size is 8.00
parent 1 is file 7

attribute # 10 is a key, name of att: J#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	1.00

attribute # 12, name of att: jcity,
attribute size is 6 bytes.

value	# of occurrences
athens	1.00

At this point, the relations also contain attributes that were needed for the selections, but are not required for the final answer to the query. Those attributes are removed and we are left with attribute numbers 1, 2, 4, 5, 8, 10, and 13.

LOG FILE AFTER 2ND PROJECTION

* * * * * *

file # 5 at site # 1 has 5.0 tuples, file size is 65.00,
parent 1 is file 1

attribute # 1 is a key, name of att: s#, has 5 possible values, attribute size 2 bytes.

value	# of occurrences
s#	5.00

attribute # 2, name of att: sname, has 5 possible values, attribute size is 5 bytes.

value	# of occurrences
smith	1.00
jones	1.00
blake	1.00
clark	1.00
adams	1.00

attribute # 4, name of att: scity,
attribute size is 6 bytes.

value	# of occurrences
london	2.00
paris	2.00
athens	1.00

*　　*　　*　　*　　*　　*

file # 9 at site # 3, has 6.0 tuples, file size is 54.00, parent 1 is file 4.

attribute # 1 is a key, name of att: s#, has 5 possible values, attribute size is 2 bytes.

value	# of occurrences
s#	6.00

attribute # 5 is a key, name of att: p#, has 6 possible values, attribute size is 2 bytes

value	# of occurrences
p#	6.00

attribute # 10 is a key, name of att: j#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	6.00

attribute # 13, name of att: qty,
attribute size is 3 bytes.

value	# of occurrences
400	2.00
500	1.00
600	1.00
700	1.00
800	1.00

*　　*　　*　　*　　*　　*

file # 11 at site # 2, has 2.0 tuples, file size is 8.00 bytes, parent 1 is file 8

attribute # 5 is a key, name of att: p#, has 6 possible values, attribute size is 2 bytes.

value	# of occurrences
p#	2.00

attribute # 8, name of att: weight, has 30 possible values, attribute size is 2 bytes.

value	# of occurrences
12	0.67
14	0.33
17	0.67
19	0.33

* * * * * *

file # 12 at site # 3, has 1.0 tuples, file size is 2.00 bytes, parent 1 is 10.

attribute # 10 is a key, name of att: J#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	1.00

After performing all of the possible projections and selections, the relations are ready to be joined. The 4 relations which contain relevant information are : 5, 9, 11, and 12. The linear programming method is used to select two files out the possible four. The LP algorithm found that it is best to perform a local join at site 3 between files 9 and 12. As a result one gets a new file, 13. The cost of this join is found to be 23800 monetary units.

LOG FILE AFTER JOINING FILES 9 AND 12

* * * * * *

file # 5 at site # 1, has 5.0 tuples, file size is 65.00 parent 1 is 1

attribute # 1 is a key, name of att: s#, has 5 possible values, attribute size is 2 bytes.

value	# of occurrences
s#	5.00

attribute # 2, name of att: sname, attribute size is 5 bytes.

value # of occurrences

```
 smith                     1.00
 jones                     1.00
 blake                     1.00
 clark                     1.00
 adams                     1.00

attribute # 4, name of att: scity,
attribute size is 6 bytes.

value               # of occurrences
 london                    2.00
 paris                     2.00
 athens                    1.00
```

*   *   *   *   *   *

```
file # 11  at site # 2 has 2.0 tuples, file size is  8.00
bytes parent 1 is  8


attribute #  5 is a key,  name of att: p#, has 6 possible
values, attribute size is 2 bytes.

value               # of occurrences
 p#                        2.00

attribute #  8, name of att: weight,
attribute size is 2 bytes.

value     # of occurrences
  12         0.67
  14         0.33
  17         0.67
  19         0.33
```

*   *   *   *   *   *

```
file # 13  at site # 3, has 0.9 tuples, file size is  7.71
parent 1 is file 12   parent 2 is  file 9


attribute # 10 is a key, name of att: J#, has 7 possible
values, attribute size is 2 bytes.

value               # of occurrences
 j#                        0.86

attribute # 1 is a key, name of att: s#, has 5 possible
values, attribute size is 2 bytes.
```

value # of occurrences
s# 0.86

attribute # 5 is a key, name of att: p#, has 6 possible values, attribute size is 2 bytes.

value # of occurrences
p# 0.86

attribute # 13, name of att: qty, attribute size is 3 bytes.

value	# of occurrences
400	0.29
500	0.14
600	0.14
700	0.14
800	0.14

Now there are 3 relations in the database, the LP algorithm has choosen to send relation 13 to site 2 to be joined with relation 11. As a consequence, a new relation, 14, is created. The cost of this join is 7050 monetary units.

LOG FILE AFTER JOINING FILE 13 and 11

* * * * * *

file # 5 at site # 1, has 5.0 tuples, file size is 65.00 bytes, parent 1 is file 1

attribute # 1 is a key, name of att: s#, has 5 possible values, attribute size is 2 bytes

value # of occurrences
s# 5.00

attribute # 2, name of att: sname, attribute size is 5 bytes.

value	# of occurrences
smith	1.00
jones	1.00
blake	1.00
clark	1.00

adams 1.00

attribute # 4, name of att: scity,
attribute size is 6 bytes.

value	# of occurrences
london	2.00
paris	2.00
athens	1.00

* * * * * *

file # 14 at site # 2, has 0.3 tuples, file size is 3.14 bytes, parent 1 is file 13 parent 2 is file 11.

attribute # 5 is a key, name of att: p#, has 6 possible values, attribute size is 2 bytes.

value	# of occurrences
p#	0.29

attribute # 8, name of att: weight,
attribute size is 2 bytes.

value	# of occurrences
12	0.10
14	0.05
17	0.10
19	0.05

attribute # 10 is a key, name of att: J#, has 7 possible values, attribute size is 2 bytes.

value	# of occurrences
j#	0.29

attribute # 1 is a key, name of att: s#, has 5 possible values, attribute size is 2 bytes.

value	# of occurrences
s#	0.29

attribute # 13, name of att: qty,
attribute size is 3 bytes.

value	# of occurrences
400	0.10
500	0.05

```
 600          0.05
 700          0.05
 800          0.05
```

The last join is performed between files 14 and 5. The LP algorithm has found that it is most economic to send file 14 to site 1 to be joined with file 5. The cost of this join is 25695 monetary units.

LOG FILE AFTER JOINING FILE 14 AND FILE 5

*   *   *   *   *   *

```
file # 15 at  site # 1, has 0.3 tuples, file size is  6.29
bytes, parent 1 is file 14    parent 2 is  file 5


attribute #  5 is a key, name of att: p#, has 6 possible
values, attribute size is 2 bytes.

value                 # of occurrences
 p#                          0.29

attribute # 8, name of att: weight,
attribute size is 2 bytes.

value      # of occurrences
  12          0.10
  14          0.05
  17          0.10
  19          0.05

attribute # 10 is a key, name of att: J#, has 7 possible
values, attribute size is 2 bytes.

value                 # of occurrences
 j#                          0.29

attribute #  1 is a key, name of att: s#, has 5 possible
values, attribute size is 2 bytes.

value                 # of occurrences
 s#                          0.29

attribute # 13, name of att: qty,
attribute size is 3 bytes.

value     # of occurrences
 400         0.10
```

500	0.05
600	0.05
700	0.05
800	0.05

attribute # 2, name of att: sname,
attribute size is 5 bytes.

value	# of occurrences
smith	0.06
jones	0.06
blake	0.06
clark	0.06
adams	0.06

attribute # 4, name of att: scity,
attribute size is 6 bytes.

value	# of occurrences
london	0.11
paris	0.11
athens	0.06

File 15 contains key attributes which are not required by the query. These attributes are now removed and we are left with attributes 2, 4, 8, and 13.

LOG FILE AFTER 3rd PROJECTION

*　　*　　*　　*　　*　　*

file # 16 at site # 1 has 0.3 tuples, file size is 4.57
bytes, parent 1 is file 15

attribute # 8, name of att: weight,
attribute size is 2 bytes.

value	# of occurrences
12	0.10
14	0.05
17	0.10
19	0.05

attribute # 13, name of att: qty,
attribute size is 3 bytes.

value	# of occurrences
400	0.10
500	0.05
600	0.05
700	0.05
800	0.05

attribute # 2, name of att: sname,
attribute size is 5 bytes.

value	# of occurrences
smith	0.06
jones	0.06
blake	0.06
clark	0.06
adams	0.06

attribute # 4, name of att: scity,
attribute size is 6 bytes.

value	# of occurrences
london	0.11
paris	0.11
athens	0.06