A QUERY OPTIMIZATION TECHNIQUE IN RELATIONAL DATABASES

By

FEROZE KHALIFULLAH

Bachelor of Engineering

P.S.G. College of Technology

Coimbatore, India

1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1991

Thesis
1991
K445g

Oklahoma State Univ. Library

A QUERY OPTIMIZATION TECHNIQUE

IN RELATIONAL DATABASES

Thesis Approved:

Huizhu Lu

Thesis Advisor

William David Miller

M. Samadzadeh-H.

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my appreciation to Dr. Huizhu Lu, for her encouragement, advice, and guidance. I also wish to thank my committee members, Dr. Mansur H. Samadzadeh and Mr. William D. Miller for their assistance.

Additionally, I wish to thank my parents and my friends for encouraging and supporting me throughout my graduate program, and to all other individuals who helped me in this thesis and during my coursework at Oklahoma State University.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

Motivation 6
Outline of the thesis 7

II. LITERATURE REVIEW AND BACKGROUND 8

Query Processing in Centralized Databases 10
Query Processing in Distributed Databases 13
Transaction Management 16
Storage Structure 17

III. QUERY PROCESSING 19

Definitions 19
Optimization Objectives 22
Query Representation 23
Optimization Techniques 30

IV. PROPOSED APPROACH 34

Introduction 34
Target Area 36
Description of the Proposed Method 37
An Example 42
Simulation Details 45

V. RESULTS AND CONCLUSIONS 49
Evaluation Criteria 49
Results 50
Conclusions 55
Future Work 55

REFERENCES 57

APPENDIX - PROGRAM IMPLEMENTATION 61

LIST OF TABLES

Table Page

I. Comparison of the number of disk accesses made between the two methods when the number of attributes queried = 3 51

II. Comparison of the number of disk accesses made between the two methods when the number of attributes queried = 4 51

III. Comparison of the number of disk accesses made between the two methods when the number of attributes queried = 5 52

LIST OF FIGURES

Figure Page

1. A Sample Database System, consisting of three relations S, P, and SP 20

2. Projection of the attributes SNAME and SCITY on relation S 26

3. Selection of those attributes in relation SP where QTY > 300 27

4. An Equijoin on relations S and SP with the predicate S.S# = SP.S# 29

5. A Natural Join on the relations S and SP with the predicate S.S# = SP.S# 30

6. Pseudo-code for Nested Loop Join 31

7. Temporary Relation SP' produced when the queries are processed separately 42

8. Temporary Relation SP' produced when the queries are processed together 44

9. Pseudo-code for the Simulation of the Proposed Query Processing Approach 47

10. Graph showing the relation between the # of common selection attributes and the percentage reduction in disk accesses when # of attributes queried = 3 53

11. Graph showing the relation between the # of common selection attributes and the percentage reduction in disk accesses when # of attributes queried = 4 53

12. Graph showing the relation between the # of common selection attributes and the percentage reduction in disk accesses when # of attributes queried = 5 54

CHAPTER I

INTRODUCTION

A database system is basically a computerized record-keeping system - that is, a system whose overall purpose is to maintain information and make that information available on demand [Date87]. Database Management systems (DBMSs) are now used in almost every computing environment to organize, create and maintain important collections of information. The information concerned can be anything that is deemed to be of significance to the individual or organization who maintains the database. It is something like an electronic filing cabinet. The user of the system will be able to perform a variety of operations on the database such as:

i. adding new data,

ii. deleting existing data,

iii. retrieving data (and changing if needed).

Database systems are now available on machines that range from small micros to the large mainframes. The data in a database is accessed using queries and the high-level computer language that is used to access this data is known as a query language. The hardware for handling very large data sets is now available. However these large data sets need careful handling. Many early Database Management

Systems (DBMSs) have been criticized because of their inefficient way of handling the powerful operations they offer, particularly the content-based access to data by queries. Accessing the data in the wrong way can slow down the computer for a considerable time resulting in waste of resources and time.

Depending on the approach used to construct the database management systems, database systems can be broadly divided into two types, namely nonrelational systems and relational systems.

A majority of present-day database systems are relational [Date87]. The database stores all information in tables and can manage data by direct manipulation of these tables without reference to other constructs. User queries specify what information is wanted but not how this information is to be extracted from the database. These nonprocedural languages are easier to use than the navigation languages of IMS (Information Management System) or CODASYL (Conference on Data Systems Languages). The user does not see the access plan that is used. This is where query optimizers come into the picture. The query optimizer chooses an efficient access path for the query using information about the structure of the database. It translates the predicate specification into an algorithm to perform database access to solve the query.

The advantage of optimization is twofold. Firstly, the user is not concerned with how best to state the query (as

compared to navigational databases), and secondly, there is a high possibility that the optimizer might actually do better than a human programmer. This can be illustrated by the following example.

Consider a database with two relations, S and SP, as shown in the below. Let us assume that relation S has 100 entries and SP has 10,000 entries.

S

S#	SNAME	STATUS	CITY

SP

S#	P#	QTY

Now if we have a query such as "Who are the suppliers who supply part P3?", then one solution (the worst possible) is to create a temporary table that has information from all tables and pick out the rows that meet all the conditions. If only 50 shipments were made for part P3, then in our case we would have to read 10,100 tuples and construct a temporary relation which is the Cartesian product of the relations S and SP. And now, using the where clause, we would pick out 50 tuples and then project this result over relation S, to obtain the desired result (which would contain 50 tuples at the most).

An alternative method (that could be used by the query optimizer) would be to restrict relation SP to those tuples that contain part P3. This would involve reading 10,000 tuples but producing a result of 50 tuples from them. Now if we join this result to relation S, then we would have a

retrieval of only 100 tuples and the result would contain only 50 tuples. Subsequent projection (as in the previous case) would yield a final result of 50 tuples (at the most). Now if the number of disk writes were to be taken as a performance measure, then obviously the latter method far outperforms the former. This contrived example illustrates the necessity for optimization in query processing.

The overall purpose of the optimizer, then, is to choose an efficient strategy for evaluating a given relational expression. Query optimization tries to solve this problem by integrating a large number of techniques and strategies, ranging from logical transformations of queries to the optimization of access paths and the storage of data on the file system level [Jar84].

Typically, in a modern relational database system, optimization involves a cost model specific to the system [Yu84]. The optimizing methods either should attempt to maximize the output for a given number of resources or to minimize the resource usage for a given output. If response time is the criterion for the performance measure, then this should be minimized.

Nonrelational systems are at a lower level of abstraction than relational systems. They include:

i. the hierarchic model,

ii. the network model,

iii. the inverted list,

iv. deductive DB's, and

v. object-oriented DB's.

But the major nonrelational approaches to database are the hierarchic and network approaches. The hierarchic system, exemplified by IBM's (IBM is a trademark of International Business Machines Corporation) IMS, has a data model that requires all data records to be assembled into a collection of trees. Each individual tree in the database can be regarded as a subtree of a hypothetical "system" root record i.e the entire database can be considered as a single tree. The notion of hierarchic sequence defines a total ordering for the set of all records in the database, and the database is regarded as being logically stored in accordance with that ordering. The query language permitted an application programmer to access one record at a time, starting from a specific entry point record and moving on to the desired information.

In the network database system, the collection of the records is arranged into a directed graph as typified by the CODASYL standards. It consists of a set of records and a set of links. The main distinction with the previous model is that in the hierarchic structure, a child record has exactly one parent record whereas in a network structure a child record can have any number (even zero) of parents.

Both the tree-based and graph-based database systems, needed a navigational query language defined. Due to their record-at-a-time orientation, these systems are all fundamentally programming systems [Date87]. To answer a

specific database request, an application programmer, skilled in disk-oriented optimization, must write a complex program to navigate through the database i.e the user must tell the database manager precisely how to find the desired information. Any 'optimization' is generally performed by the user, not by the system. These are therefore efficient in the hands of an expert, but in general are harder to use than relational database managers. The majority of users of large databases are novice programers and the restrictions that apply to network databases still hold [Rob88]. Moreover when the structure of the database changes, as it will whenever new kinds of information are added, application programs usually need to be rewritten. These database systems are costly to use because of the low-level interface between the application program and the DBMS [Silb91].

Motivation

A number of algorithms to process queries in different distributed and centralized database systems have been proposed and implemented [Yu84]. But, as mentioned earlier, no particular method is suitable for all environments and there is room for still further advancement in query optimization. This is the main motivation for working on this topic. Primarily, the problem of optimizing queries in centralized DBMSs will be considered in this thesis, because the cost criterion of disk access is taken as the main criterion for performance measure. Moreover centralized

query optimization is not only important in many mainframe databases, and more recently in microcomputer DBMSs - but it also appears as a subproblem of query optimization in distributed database systems.

Outline of the thesis

In chapter II, a brief outline of the various approaches towards query optimization is given. In chapter III, the working of query optimization techniques are described, with special emphasis to relational algebra concepts, as this is the data manipulation method used in this thesis. In chapter IV the details of the proposed approach to query optimization along with the simulation details of this approach as run on the Ultrix-32 based VAX 8350 are given. Finally in chapter V, the results, limitations, conclusions, and scope for future work are outlined.

CHAPTER II

LITERATURE REVIEW AND BACKGROUND

In this section the database literature on query optimization and closely related work is reviewed. There are several aspects to query optimization. One aspect involves the type of database under consideration (distributed or centralized), another aspect concerns the type of optimization and yet another involves the strategy adopted.

A primary goal of a DBMS is to insulate users and programs from the physical structure of the data, and this is accomplished by high-level query languages such as SQL (Structured Query Language) by translating a logical query into a sequence of operations at the physical level. The problem of generating reasonable natural language-like responses to queries, formulated in non-navigational query languages with logical data independence, is addressed by Wald and Sorenson [Wald90]. But sometimes a conceptual query can be translated into more than one logical query, and they also address the problem of ambiguity in a formal query language. The sizes of derived relations often play an important role in selecting a plan for evaluating a query, whether it is a centralized database or a distributed database. Answering queries in a relational database often

requires that the natural join of two or more relations be computed. Join operations are the most time-consuming relational operations, since their result can have a size equal to the product of the sizes of the original relations. In this regard the effect of join operations on relation sizes is discussed by Gardy and Puech [Gar89]. They present a model of relations and show how to use it to deduce probabilistic estimations of derived relation sizes. The result of a join operation does not always have the expected value (i.e. not all joins are lossless joins). This is discussed by Aho et al [Aho79] with emphasis on whether the join of several relations is 'lossless' or 'lossy'. De Vet [Vet89] described an algorithm for evaluating database queries represented as expressions in a logical language. The algorithm described in the paper evaluates a query expression recursively in terms of its subexpressions. It can interrupt input expressions which are internally structured as trees or directed acyclic graphs.

A user may not always have a choice of what query language to use but sometimes his or her choice may span over a few prespecified alternatives. Jarke and Vassiliou [Jar85] gave a framework for choosing a database query language. A methodology for selecting a type of query language interface on the basis of its functionality and the "user friendliness" characteristics is discussed. They developed taxonomies of query languages and language users. Their approach to query language evaluation can be

understood as a specialized cost-benefit analysis method, in the sense that multiple evaluation criteria are based on a simple economic model of query language usage and the trade-off between costs and benefits depends on the user type. This immediately leads to a two-level classification of query language: by the senses employed by the user and by the language methods. Now this classification serves as a tool for evaluating query languages in a structured manner.

Query Processing in Centralized Databases

A centralized database is one in which the complete data is maintained at one location. A wide variety of approaches to improve the performance of query evaluation algorithms in centralized databases have been proposed: such as logic-based and semantic transformations, fast implementations of basic operations, and heuristic algorithms for generating alternative access plans and choosing among them. Jarke and Koch [Jar84] present a survey of query optimization techniques using the relational calculus representation of queries. Their work is divided into four steps namely:

i. query representation,

ii. query transformation,

iii. query evaluation, and

iv. access plans.

Techniques for representing queries in terms of their suitability for optimization are compared. After being

transformed, a query must be mapped into a sequence of operations that return the requested data, and the implementation of such operations on a low-level system of stored data and access paths are also analyzed. In addition, nonstandard query optimization issues such as higher level query evaluation, query optimization in distributed databases, and the use of database machines are also addressed. But the focus is on query optimization in centralized databases.

Decomposition as a strategy for query processing in the database management system INGRES is addressed by Wong and Youssefi [Wong76]. The general idea is to decompose the query into a sequence of one-variable queries by alternating between reduction and tuple substitution. Reduction means the breaking off components of the query which are joined to it by a single variable and tuple substitution means substituting for one of the variables, a tuple at a time. Algorithms for reduction and for choosing the variable to be substituted are given, and the query processing algorithm developed for QUEL (Query Language) which is the data language for the INGRES ("Interactive Graphics and Retrieval System) system is also described.

Chakravarthy et al [Cha90] discuss semantic query optimization. The idea of semantic query optimization is to transform the user query into one which is semantically equivalent to the original query and which can be processed more efficiently. There are several aspects to semantic

query optimization. One aspect involves the type of database under consideration. Another aspect concerns the generation of semantically equivalent queries, and their correctness. Yet another aspect is the pruning of information that is not useful for semantic optimization. They describe a method of semantic query optimization in deductive databases couched in first-order logic. They also show how semantic query optimization techniques can be extended to databases that support recursion and integrity constrains that contain recursion.

Sellis and Shapiro [Sel91] presented the use of database query languages for programming non-traditional applications such as engineering and artificial intelligence, and the techniques for the optimization of such programs. They primarily focused on extended query languages that include an iteration operator, but several of the techniques apply to classical query languages also. Whang and Krishnamurthy [Wha90] have presented techniques for optimizing queries in memory-resident database systems. Their approach is towards developing a CPU-intensive cost model. Therefore, the emphasis is on main-memory query processing and the effect of the operating system's scheduling algorithm on the memory residency assumption. They have used the OBE (Office-By-Example) integrated office system that has been under development at IBM Research.

Databases satisfy the needs of users with regard to business applications, but at times they must be expanded to

offer services in object management and knowledge management. Object management includes efficient storing and manipulating nontraditional data types such as icons or pictures. Knowledge management entails the ability to store and enforce a collection of rules that are part of the semantics of an application. In this regard, Stonebraker and Kemnitz [Ston91] discuss the details of a database manager that incorporates the above two additional dimensions.

Query Processing in Distributed Databases

The demand for more and more information by different organizations leads to very large databases, that will exceed the physical limitations of centralized systems. Also sometimes integration of already existing geographically dispersed databases is required. This leads to the concept of distributed databases.

A distributed database is one in which the data is maintained at multiple sites [Silb91]. The multiple sites are connected together into some kind of communications network, in which an end user or application programmer at any site can access data stored at any other site. Usually the data is maintained at the site closer to the people who are responsible for it, thereby reducing communication costs. For example information about the Texas customers of a company might be stored on a machine in Dallas, while data about Illinois customers could exist on a machine in Chicago. On top of the network, a distributed DBMS can be

built in such a way that the user need not know about the distribution of the logical and physical components of the database.

This type of database has multiple advantages, such as in the event of a crash at any one site, only part of the data is inaccessible, and this may not disrupt the normal functioning too much. Distributed database systems relieve the burden on shared mainframes. Also an open architecture distributed database system will be a big help to users who increasingly need access to more than one database. Other advantages include increased availability, decreased access time, and easy expansion [Aper88].

The acceptance and widespread usage of distributed databases will highly depend on their efficiency. In this regard, Stonbraker [Ston89] mentions the desired features of a distributed database in terms of the seven transparency features, namely:

i. location transparency,
ii. performance transparency,
iii. copy transparency,
iv. transaction transparency,
v. fragment transparency,
vi. schema change transparency, and
vii. local DBMS transparency.

Various techniques for optimizing queries in distributed databases are presented by Yu and Chang [Yu84]. They outline the main points of the ideas extracted from existing

algorithms such as: the use of semi-join to reduce transmission cost, the characteriation of queries solvable by semijoins, the transformation of cyclic queries into tree queries, enhancements of semijoin strategies, the optimal processing of certain restricted types of queries, etc. The problem of allocating the data of a database to the site of a communication network is investigated by Peter M. G. Apers [Aper88]. A model that makes it possible to compare the cost of allocation is presented by him.

A query processing algorithm for distributed relational database systems is given by Egyhazy and Triantis [Egy88]. A new technique is proposed to compute the resulting relation size after a projection, a selection, and a join on non-key attributes (or key attributes). This is then used in the algorithm to determine which relation is to be sent across the network for processing.

PRECI* (Prototype of a RElational Canonical Interface) is a research prototype for a generalized distributed DBMS. The architecture of PRECI* is explained by Deen et al [Deen85]. The architecture presented has the ability to provide location transparency and transaction-oriented queries, the former providing for data integration and meta data. Similarly R* is an experimental, distributed DBMS developed and operational at the IBM Almaden Research Center. In a large organization, it is difficult to standardize on a single DBMS since the requirements of the DBMS from different units (such as engineering,

administrative etc.) are diverse. Under these circumstances, the heterogeneous DBMS is an effective way of sharing data. Chin-Wan Chung describes the architecture of DATAPLEX, which is a heterogeneous distributed DBMS developed at General Motors Research Laboratories.

Transaction Management

A transaction is a sequence of operations that must appear "atomic" when executed. For example when a student drops course A and adds course B, then the database system should ensure that either both of the operations - drop A and add B- happen or that neither happens. If only one of them happens, then an inconsistent database state results. A transaction must transform the database from one consistent state to another. For this, concurrent transactions in the system must be synchronized correctly in order to guarantee that consistency is preserved. For example system failures must not result in an inconsistent state. In a distributed database system, the actions of a transaction may occur at more than one site, hence this problem is felt more in these systems. In this regard Mohan et al [Moh86] discuss transaction management in the R* DBMS. They also discuss R*'s approach toward distributed deadlock detection and resolution.

Storage Structure

A query optimization algorithm has to choose among a variety of existing access paths to resolve a query. In this regard the internal details of implementing such access paths are important. This depends on the file structures used. The internal level of the database is the level that is concerned with the way the data is actually stored on the disk. Usually databases are physically stored on direct access devices, like moving-head disks etc. Disk access times are much slower than main storage access time. Therefore minimization of the number of disk accesses is an important objective in the optimization process. Now optimization can also be influenced by the storage structure i.e the way the data is arranged on the disk, and related access strategies.

Indexing is a method of speeding up retrievals. An index is a special kind of stored file, in which each entry contains two values: a data value, and an associated pointer. Usually the index is maintained on the primary key (i.e the identifying key for the tuple), and the pointer identifies a record of that file that has the same value in its primary key attribute as the data value of the index. But the main disadvantages of indexing is that it slows down updates, and it takes up considerable overhead. Moreover indexes cannot be maintained for every attribute in the relation and there is no saying as to which attribute will be queried by the user.

Hashing is another method of direct access to the records. Each record has its hash address as a function of the primary key of the record. But hashing has its own limitations, as it leads to physical clustering and also the possibility of running into collisions - that is two distinct records that hash to the same address. Moreover as the size of the hash file increases, the number of collisions also tends to increase [Date87], and hence the average access time increases correspondingly. In this regard extendable hashing is a nice variation on this base technique, since extendible hashing guarantees that the number of accesses to a particular record is never more than two (usually one).

Compression techniques are also used to reduce the amount of storage space required for a given collection of data. Where the data is accessed sequentially or accessed by a single index, differential compression can be used. Huffman coding is a character encoding technique, that can result in significant data compression if different characters occur in the data with different frequencies.

CHAPTER III

QUERY PROCESSING

Definitions

Since this work is primarily concerned with relational databases, at the outset, it would be appropriate to explain the related terms in a clear and precise manner.

Relational Database

According to Date [Date87] "a relational database system is a database that is perceived by its users as a collection of tables (and nothing but tables)". The data is not physically stored as tables, but at the internal level the system is free to use any structure it pleases, provided that it is capable to represent those structures as relations at the higher levels. The principles of the relational model were laid down by Dr. E.F.Codd [Codd70], the major features of which are:

i. Each relation contains only one record type.

ii. The records have no particular order.

iii. The records have a unique identifier field or field combination called the primary key.

iv. Every attribute is single valued.

For example, the database in Figure 1 consists of three relations. These databases do not allow repeating groups and the entire information content of the database is represented as explicit data values. In contrast to nonrelational systems, the relationship between two tuples in two different tables is not represented by any kind of physical link (such as pointer). Instead it is expressed by the presence of a tuple in another table (say table T) having the primary key values of the two related tuples in the same row of table T.

S

S#	SNAME	STATUS	SCITY
s1	Smith	20	Dallas
s2	Adams	10	OKCity
s3	James	15	Austin
s4	Tony	23	Waco
s5	Mike	40	Tulsa

P

P#	PNAME	PCITY
p1	Rim	Tulsa
p2	Drum	NYCity
p3	Hub	Norman
p4	Disk	Boston

SP

S#	P#	QTY
s1	p1	100
s1	p2	180
s1	p4	50
s2	p1	200
s3	p3	400
s3	p4	80
s4	p1	800
s4	p2	200
s4	p3	150
s4	p4	300
s5	p2	200
s5	p4	300

Figure 1. A sample relational database system, consisting of three relations, S, P, and SP

Relational database systems have high-level query languages to ease the use of the DBMS by both end users and application programmers. The application programmer needs to

only specify the predicate(s) that identifies the desired record or combination of records from the database. Also the theory of normalization was formulated to help with database design by eliminating redundancies and certain logical anomalies.

Queries

A query is a language expression that describes data to be retrieved from a database [Jar84]. Queries are used in several settings, the most obvious application is that of direct requests by end users who need information from the database (e.g., "What are the names and the grade in course A of the students who are enrolled in course B, and are TA's?"). Another application of queries occurs in transactions that change the stored data based on their current value (e.g., "give all administrative staff, working for the biology department a 10 percent salary increase").

Optimization

Optimization in database systems means to choose among the various ways of executing a single query [Sel91]. Query optimization tries to minimize the response time for a given query language and mix of query types in a given system environment. In relational databases the query (in the high-level relational query language) is translated into plans that are as efficient (if not more) as what a skilled

programmer would have written using one of the earlier DBMSs (such as network or hierarchical) for accessing the data.

Optimization Objectives

There are two main objectives of query optimization. These are:

i. to minimize the response time, and

ii. cost minimization of technical usage.

The response time goal is reasonable only under the assumption that the user time is the most important criteria. Otherwise, minimization of the resource usage for a given output is the main objective. Fortunately, both the above mentioned objectives are largely complementary. But sometimes when conflicts arise, they are resolved by assigning limits to the availability of technical resources.

The total cost to be minimized is the sum of the following [Jar84]:

i. Secondary storage access cost,

ii. Storage cost,

iii. Computation cost, and

iv. Communication cost.

Secondary storage access cost is the cost associated with the loading of data pages from secondary storage into memory. The factors that influence is the amount (or number) of data to be retrieved, the way the data is clustered in the physical storage, the size of the memory buffer, and the speed of the devices used. This cost is usually measured by

the number of page accesses made. Storage cost are relevant only if storage becomes a system bottleneck, and the storage can be varied from query to query. This is the cost associated with keeping the storage buffers used over a period of time. Computation cost is the cost associated with using the central processing unit (CPU) engaged while computing the access strategy. Communication costs are expressed in terms of the total amount of data transmitted.

The type of query optimization sought is strongly influenced by the trade-offs among the above cost components. For example, in distributed DBMS, communication costs are relevant but not so much so in centralized databases. In the latter, the cost is dominated by the secondary storage access time. In such systems communication costs are independent of the query evaluating strategy. Since the focus of this thesis is on centralized databases, communication costs are not considered.

Query Representation

Queries can be represented in a number of forms. In the context of query optimization, the query representation must be powerful enough to express a large class of queries [Jar84]. The relational database model can be expressed in the following models:

i. relational calculus,

ii. relational algebra,

iii. query graphs, and

iv. tableaus.

This constitutes the data manipulation part. The manipulative part provides a set of operators for data manipulation. These are not just for data retrieval but they form the basis on which optimization techniques are founded. Most of the work done on query optimization is in the framework of the relational model where the queries are represented as relational calculus or relational algebra expressions [Vet89].

Relational Calculus

The relational calculus provides a notation for defining the result of a query through a description of its properties. The representation of the query consists of two parts namely the target list and the selection expression. The target list defines the attributes that appear in the result and the selection expression specifies the contents of the relation resulting from the query. Using the sample database in Figure 1, the query "Get supplier names and supplier city name for suppliers who supply part P4", may have a calculus representation something similar to as shown:

> Get SNAME and SCITY from relation S, such that there exists a tuple in relation SP with the same S# and P# value 'P4'.

The user just defines the characteristics of the desired set, and it is left to the system to decide exactly what

projections, joins or selections etc. must be executed. In the target list of the above query, the range variable is 'SNAME' and 'SCITY' and they 'range over' relation S.

Relational Algebra

The relational algebra consists of a set of operators (such as join, project, etc.) defined on relations. These operators fall into two sets namely:

i. the traditional set operations, such as union, intersection, difference and Cartesian product.

ii. the special relational operations select, project, join, and divide.

Each operator of the relational algebra takes either one or two relations as its input and produces a new relation as its output (which may even be an empty relation). As opposed to relational calculus expression, which describes the relation resulting from a query by means of its properties, a relational algebra expression provides a collection of explicit operations such as join, projection, selection etc. - that can be used to build the desired relation from the given relations of the database. The calculus simply states the problem but the algebra gives a method of solving that problem. For example the relational algebra expression of the example query in the previous section may be like this:

Join relation S with relation SP.

Restrict the result to those tuples that have P# = P4.

Project the result got on SNAME and SCITY.
However relational calculus and relational algebra are equivalent to one another [Date87]. In this thesis relational algebra is used in the optimization process. In particular the three basic relational operators, 'Join', 'Project', and 'Select' are used. So the underlying section explains the process of join, project and select as applied on the relations.

Project

This operator extracts specified attributes from the given relation. It takes one relation as its input and always builds a relation that is non-empty. It yields a 'vertical' subset of the input relation. In the process it eliminates redundant duplicate tuples that may exist after the projection.

S

S#	SNAME	STATUS	SCITY
s1	Smith	20	Dallas
s2	Adams	10	OKCity
s3	James	15	Austin
s4	Tony	23	Waco
s5	Mike	40	Tulsa

SNAME	SCITY
Smith	Dallas
Adams	OKCity
James	Austin
Tony	Waco
Mike	Tulsa

Figure 2. Projection of the attributes SNAME, and SCITY on relation S

Any number of attributes may be projected but the attributes of the result will have the same qualified names as they have within the original relation from which the projection

was done. A projection of the attributes "S#", and "STATUS" on the relation S of Figure 1 is shown in Figure 2.

Select

Let *theta* represent a valid scalar comparison operator (such as =, <, >=, !=, etc. The selection of those attributes in relation R, whose A-attribute *theta* a specified constant, is denoted by

R where R.A *theta* specified constant,

where A is an element of R. Now this is obtained by choosing those rows in R such that R.A *theta* specified constant evaluates to be true. In short selection produces a 'horizontal' subset of the given relation. The result may contain 0 or more tuples (unlike projection).

Selection operation: SP where QTY > 200

SP

S#	P#	QTY
s1	p1	100
s1	p2	180
s1	p4	50
s2	p1	200
s3	p3	400
s3	p4	80
s4	p1	800
s4	p2	200
s4	p3	150
s4	p4	300
s5	p2	200
s5	p4	300

S#	P#	QTY
s3	p3	400
s4	p1	800
s4	p4	300
s5	p4	300

Figure 3. Selection of those attributes in relation SP where QTY value > 200

One or more select clauses on the same relation may be used in selection. The selection operation not only permits simple comparison in the WHERE clause but by the property of closure it is possible to extend this (making use of other constructs such as UNION, MINUS etc.). A simple example of a selection criteria as applied on the relation SPJ of Figure 1 is shown in Figure 3.

Join

Most relational databases create temporary tables to combine information from pairs of tables. Joins are a way of combining two tables. A join of two relations is a subset of their Cartesian cross product. Let *theta* be defined as any valid scalar comparison operator. Then the theta-join of relation R1 on attribute A1 with relation R2 on attribute A2, is the set of all concatenated tuples of R1 and R2 such that the condition R1.A1 *theta* R2.A2 holds good, for the respective tuples of R1 and R2. Attributes R1.A1 and R2.A2 should be defined on the same domain and *theta* must make sense for that domain.

For example an equal-to join (also called *equijoin*) on the relations S and SP of Figure 1 is given in Figure 4. It is obvious that the result of a equijoin must include two identical attributes. If one of these attributes is eliminated then the join is called a natural join i.e a natural join is the projection of a restriction of a product. Natural join is the most important form of join in

practice and it is used extensively in this work too. For simplicity the natural join of the above example can be written as:

S join SP

(S times SP) where S.S# = SP.S#

S#	SNAME	STATUS	SCITY	S#	P#	QTY
s1	Smith	20	Dallas	s1	p1	100
s1	Smith	20	Dallas	s1	p2	180
s1	Smith	20	Dallas	s1	p4	50
s2	Adams	10	OKCity	s2	p1	200
s3	James	15	Austin	s3	p3	400
s3	James	15	Austin	s3	p4	80
s4	Tony	23	Waco	s4	p1	800
s4	Tony	23	Waco	s4	p2	200
s4	Tony	23	Waco	s4	p3	150
s4	Tony	23	Waco	s4	p4	300
s5	Mike	40	Tulsa	s5	p2	200
s5	Mike	40	Tulsa	s5	p4	300

Figure 4. An equijoin on relations S and SP with the predicate S.S# = SP.S#

Now if there are more than one attribute in common between the two relations then all the common attributes in both the relations should hold the same value before the concatenation of those tuples can take place.

The join construct is associative i.e a sequence of joins can be written without any parentheses. Therefore

(R1 join R2) join R3 is the same as

R1 join (R2 join R3), which is the same as

R1 join R2 join R3.

S#	SNAME	STATUS	SCITY	P#	QTY
s1	Smith	20	Dallas	p1	100
s1	Smith	20	Dallas	p2	180
s1	Smith	20	Dallas	p4	50
s2	Adams	10	OKCity	p1	200
s3	James	15	Austin	p3	400
s3	James	15	Austin	p4	80
s4	Tony	23	Waco	p1	800
s4	Tony	23	Waco	p2	200
s4	Tony	23	Waco	p3	150
s4	Tony	23	Waco	p4	300
s5	Mike	40	Tulsa	p2	200
s5	Mike	40	Tulsa	p4	300

Figure 5. A natural join on relations S and SP with the predicate S.S# = SP.S#

Optimization Techniques

Optimizing techniques can be enforced in the compiler design itself. In the compiler level optimization is done in two areas namely:

i. loop optimization, and

ii. temporary storage management.

The former relates to time optimization and the latter to space optimization. Joins take up most time (for a join of two relations the cost is approximately equal to the cost of between 5 and 20 database retrievals [Date87]) and these are implemented in the following two ways:

i. the nested loop method, and

ii. the sort/merge method.

In the nested join (R1 join R2) join R3, it is not necessary to compute the join of R1 and R2 in its entirety before computing the join of the result and R3. Instead as soon as any two tuples of R1 and R2 have been joined, it can be passed to the process that joins such tuples with the tuples of R3. The nested loop method which is used to implement the join in the proposed approach, is described by the pseudo code in Figure 6.

```
for i = 1 to num-of-tuples in R1
     scan tuple i in R1;
     for k = 1 to num-of-tuples in R2;
          scan tuple k in R2;
          check whether tuple i matches tuple k;
          if so construct concatenated tuple;
     end;
end;
```

Figure 6. Pseudo-code for nested-loop join

The sort/merge method is a method of implementing the join operation in system R database system. It sorts the two relations (which are to be joined) on basis of the values of the joining attributes, and then on this it applies the nested loop algorithm (mentioned previously).

For a given set of relations the optimizer chooses a pair to be joined first, and then the third relation to be joined to the result of the first join, and so on. This is a strategy followed by the optimizer to reduce the search space. When two relations are to be joined each having some

restrictions on them and/or projections on them, the optimizer treats it as shown below:

(R1 join R2) where restriction on R2

is treated as

R1 join (R2 where restriction on R2).

Temporary storage management in the context of database operations is how to optimize commands by reusing temporary relations. A good caching scheme can be built where temporary results are stored so that they can be used later in the execution of later commands. But the temporary table that is built in the first place should be of use to later queries. This is dealt with greater detail in the succeeding section.

Other techniques of query optimization include:

i. early restrictions, and

ii. combining operations.

It is usually advantageous to restrict the size of the relations involved in a query as early as possible. 'Select' command in SQL can be considered as a restriction, since they tend to reduce the size of the relation (in terms of the number of tuples) involved in subsequent commands. 'Project' also is a restriction. Combining operations deal with executing two or more commands at the same time, thus avoiding scanning the same tuple twice. For example a selection and a projection can be performed together. But there is no easy way to combine two different commands into

one. In cases like this changing the order of execution can increase the efficiency.

If a query is embedded in an applications program, the query can be optimized at:

i. the compile time, or

ii. the first time the query is executed, or

iii. every time the query is executed.

If the query is optimized at compile time, distribution information is not available then. When it is optimized the first time only, then it may so happen that in the middle of the program there may be some change made which affects the query processing. The ideal time would be to optimize it whenever it is executed but in that case the overhead of optimization will also be quite high.

CHAPTER IV

PROPOSED APPROACH

Introduction

As a query is made, it can be optimized. No information about other queries in the system is used. This is called single query optimization [Sel91]. The reasons for which single query optimization is done are:

i. Firstly, in business applications it is quite common that single database commands can be quite complex, so that processing each single query efficiently would result in significant improvement.

ii. Secondly, it is again common in business data processing applications to use ad hoc access to the database, in which the user issues only one query at a time, so that there is no opportunity to optimize more than one query at a time.

iii. Moreover the query processing algorithm which is most efficient is essentially an exhaustive search of all the possibilities and this complexity would grow exponentially if more than one query at a time were optimized.

But optimization techniques can be focussed on more than one query at a time. This can be done in both traditional and non-traditional applications. This works out to be quite advantageous for the following reasons:

i. The optimizer has more information on which to base its decision. For example, knowing that there will be several consecutive commands of one type, the optimizer may elect to build an index which may not be worthwhile for only one command of that type.

ii. The optimizer has more flexibility to rearrange the order and implementation of operations.

iii. In extended languages such as QUEL*, a single new database command include several traditional commands. Therefore in these languages a single query can encompass several traditional queries.

In the proposed method, this is the basic approach taken for developing enhancements to existing query processing algorithms i.e instead of processing one query at a time, two queries are processed at one time, so as to make use of some of the advantages listed above. It is usually advantageous to restrict the size of the relations involved in a query as early as possible in the execution plan. If the succeeding query is known it becomes all the more beneficial for optimization.

Target Area

The proposed method is aimed at databases in which there is high likelihood of successive queries having common attributes. Statistical databases is one such area, since they are primarily collected for statistical analysis purposes. A statistical database is a database from which aggregate information about large subsets of entities of an entity set is to be obtained [Pal87]. A database of census data is an example of a statistical database. Furthermore traditional single query optimization techniques do not provide adequate efficiency for many non business applications, such as engineering and artificial intelligence applications. Optimization techniques need to be applied to an entire program of database queries. The proposed method is therefore relevant to these areas also.

Even in business applications, there are a number of instances where two succeeding queries use the same base relations. Therefore it is useful to preserve the temporary tables that are created using joins. An example of this would be in a typical student database (maintained by universities), with queries such as:

i. what are the names of the students who passed out in the Fall of 1990 with a GPA > 3.5?

ii. what are the names of the students who passed out in the Spring of 1991 with a GPA > 3.5?

Yet another example would be:

i. What are the names of students enrolled for 9 hours or less?

ii. What are the names of students enrolled for 15 hours or more?

Thus this method can be adopted for business applications too in a limited fashion.

Description of the Proposed Method

In both the examples cited in the previous section, there were common attributes between the two queries. The proposed approach rests on this basis that successive queries have one or more common attributes in common. In the worst case if there are no attributes in common, then processing would boil down to the same time which it would take for processing a single query.

Now even though there may be common attributes between the two queries, the restrictions that are asked for in each query may not be same or even similar. If the first query was, say for example "get names of senior students with GPA > 3.5", and the second was "get names of senior students with GPA > 3.75", then the result of the first can be directly used in the second. But as shown in the second example, the restriction applied for each query may be diametrically opposite. Therefore, in the proposed method the selection criteria is a sum of both the restrictions. It uses an "OR" clause i.e during the selection process, if the

specified attribute in the tuple in the relation evaluates to 'TRUE' value for either condition 1 or condition 2, it is selected. This makes it imperative to apply another selection analysis on the resulting relation, this time only with condition 1 and once again only with condition 2. Now in total three passes are done. One of them is done on the original relation and the remaining two are done on the relation got from the result of the first operation, which is considerably smaller than the original one. In normal query processing, in total two passes would have been done but both of them would be on the same original relation.

It is also not necessary that there have to be an attribute common between two queries, before the joint processing is done. It is enough if both the queries have this much in common that they both have some attributes which belong to the same relation. The formulation of the algorithm for the proposed approach is based on the assumption that most simple queries in a relational system can be expressed by means of the relational operators projection, selection, and join. Also it is assumed that the database on which this algorithm operates does not incorporate data partitioning.

<u>Notations</u>

The permanent relations are denoted by R1, R2, etc. A1, A2, etc. are the set of attributes associated with the

relations R1, R2, etc. Temporary relations are created as a result of performing any one of the three relational operations project, select or join, and they are denoted by TR1, TR2, etc.

A query is made. PQ1 is the set of attributes requested by the query. For example, PQ1 = {a1, a2, a3,.. }. This set PQ1 is useful in projecting the final result i.e. for performing a final projection. The query also defines some operand restrictions which are placed in another set SQ1. SQ1 contains both the attributes on which the restrictions are placed, and the restricting clauses too. For example,

SQ1 = {a1 = s3, a6 != London, a4 < 300,... }

The selections will assure that the specific operand conditions have been met. Now the user is prompted for the second query. If the user decides to answer 'NO' to the prompt, the processing of the query already read in is initiated. On the other hand if the user does enter the second query, then the corresponding details for the second query are placed in sets PQ2, and SQ2.

Methodology

We begin with building a relation matrix containing the names of the relations that are used by the two queries. Next using the sets SQ1 and SQ2, a set C is built which contains the common selection attributes of query 1 and query 2, i.e.

C = SQ1 intersection SQ2.

Now for every attribute that is present in set C, the parent relation is determined. A temporary relation TRi for each common selection attribute in set C, is created using the parent relation as the original relation. This temporary relation is created such that it contains only those tuples which satisfy the conditions associated to the specified common selection attribute, in sets SQ1 and SQ2. Now in the relation matrix, the corresponding parent relation is substituted by the newly created temporary relation i.e whenever say relation R7 is to be referred to, and if this had a temporary relation created from it then the query will refer to relation TR7 instead of R7.

Now the processing of query 1 starts. First a set E is made which contains the names of all the relations that are needed for processing query 1. E is built using PQ1 and SQ1. Note that while building E, the relation matrix is referred to, and therefore E may at the start itself contain a temporary relation in place of an original one. For example

E = {R4, R7, TR3, ...}

Selections and projections are performed and new temporary relations are created. But these temporary relations are not substituted in the place of the original ones in the relation matrix. But for each relation in E for which a projection or selection (or both) was made, the new temporary relation is substituted in its place in the set E. Whenever a projection is made, the primary key attribute(s) is also projected, so as to be able to uniquely identify the

tuples in the new relation. These key attributes facilitate the join operations which follow later. This process is repeated till the set E contains only temporary relations i.e E may be something like this:

E = {TR2, TR5, ..}

Now between every temporary relation in E, which have one or more attributes in common, a natural join is made with respect to those common attributes. The result of the join is placed in the set E, and the two temporary relations that were used for the join are now removed from E. This process is also repeated till the set E contains only one relation. This is the final temporary relation created and this will contain atleast those attributes that were defined by the initial query. Using the set PQ1, the needed result attributes are projected from this temporary relation to eliminate unnecessary key attributes. This forms the answer for query 1 which is then displayed to the user.

The whole process is repeated with query 2, the only difference being that the sets PQ1 and SQ2 are used in place of PQ1 and SQ1 for building the set E. Since the relation matrix was not disturbed at all, the set E can be started with the original relations (except for those relations which had a common selection attribute). The result of query 2 is then displayed as in the former case.

An Example

An example is cited to make the explanation clearer. Consider two queries with reference to the database in Figure 1, that are to be processed:

i. Get S#, SNAME, and QTY of shipment for those suppliers who supply part P2.

ii. Get P#, PCITY, for those parts which are supplied in QTY > 300.

Normal processing would be to process query 1 followed by query 2. In this case it would involve, reading 12 tuples from the relation SP and selecting 3 from them. Then this would be followed by joining relation S with the result. It would scan 5 tuples in S and produce a result containing 3 tuples. In total the total number of page accesses (assuming each record occupies one page - an assumption which is quite possible if the records are large) would be equal to 12 + 5 = 17.

S#	P#	QTY
s1	p2	180
s4	p2	200
s5	p2	200

SP' for query 1

S#	P#	QTY
s3	p3	400
s4	p1	800

SP' for query 2

Figure 7. Temporary relation SP' produced when the queries are processed separately

Now processing the second query would have another operation of scanning 12 tuples in relation SP, producing a result containing 2 tuples. Again now scanning 4 attributes of relation P and then making a join with the result of the previous operation would end with a relation containing 3 tuples. This time the total number of pages accessed would be equal to 12 + 4 = 16. In sum for processing two queries the total number of pages accessed would be equal to 17 + 16 = 35.

Using the method proposed, the query would be processed thus:

i. Get the common attributes between the two queries. In this case it is the attribute QTY.

ii. In the relation containing the common attribute, select those tuples which fulfil either of the two restrictions, i.e either QTY > 300 or P# = p2. This involves reading 12 tuples and selecting 5 of them. This new temporary relation created is kept in the memory or cache. Let this be SP′.

iii. Now project S# and SNAME from relation S. This involves reading 5 tuples and creates a temporary relation containing 5 tuples but much smaller in size.

iv. Select from relation SP′, those tuples which meet the condition P# = p2, and produce a join with the result of the previous projection.

v. Project the attributes asked for in the result of the first query. Totally the number of page access done for this query = 12 + 5 = 17.

vi. Now project P# and PCITY from relation P. This involves reading 4 tuples and creates a new relation containing 4 tuples but again smaller in size, as compared to the original relation.

vii. Select from the relation SP′ the tuples that meet the condition, QTY > 300 and perform a join with the result of the previous step.

viii. Finally project the attributes asked for in the result of the second query, from the relation formed in the previous step. For this query the total number of pages accessed from the disk = 4, since the relation SP′ was held in the main memory or cache as the case may be.

S#	P#	QTY
s1	p2	180
s3	p3	400
s4	p1	800
s4	p2	200
s5	p2	200

New SP′ formed

Figure 8. Temporary relation SP′ produced when the queries are processed together

In total the number of page's read from the disk amounts to 17 + 4 = 21, as compared to 35 in the first case. Note also that in the relation SP there were no tuples that were commonly selected for both the queries. had there been some common tuples, then the size of the temporary relation built (i.e SP') would have been even smaller.

The overhead attached to this approach is that the temporary relation SP' which is built in the first place is larger than the one that would have been built for a single query. Also it is scanned twice.

Simulation Details

The approach towards query optimization discussed in the previous section was simulated on a VAX 8350 running under the Ultrix O/S. The coding was done in the 'C' language a copy of which is attached in the appendix.

Initially, the database containing four relations was created. The database is created of fixed length records, with fixed length fields. Each record is assumed to be big enough to occupy one complete page so that each time a record is accessed, a page access is said to be done. Each relation of the database is maintained in a separate file, and these are treated as secondary storage. The relations are created just to look like tables (the user view), and they consist of a header line followed by the tuples. No primary key value is allowed to be NULL. A separate file called 'MANAGER' is used to store information about the

database, which the algorithm reads at the beginning. This gives details about the names of the relations of the database, their primary keys, and the number of tuples in each relation.

Queries are addressed in a SQL like format with a little modification. A typical query has the syntax shown below:

Atr1 [,Atr-i] ; Condn1 [,Condn-i] .

The square parentheses '[]' denote that any number of attributes, or conditions can follow. The list of attributes is terminated with a ';' and the end of the query is indicated by a '.' The left side of the ';' in the query is read into the set PQ, and the right side into the set SQ. Both these sets are maintained by means of arrays.

The pseudo-code for the whole simulation is shown in Figure 9. The code consists of seven basic modules, namely:

i. creation of the matrix of relations,

ii. reading the input query,

iii. checking for common selection attributes,

iv. making the relation list,

v. performing the selections and projections,

vi. performing the joins, and

vii. printing the results.

```
begin
read MANAGER file;
create relation-matrix;
read 1st query;
prompt user for 2nd query;
if user sends 2nd query
     read 2nd query;
     find common selection attributes;
     create temp files for those relations only;
     adjust relation-matrix;
end;
start processing query 1
     make list of relations;
     make selections and projections;
     note # of page accesses made;
     perform joins;
     extract result;
end processing query 1;
if (2nd query present)
     make list of relations for 2nd query;
     make selections and projections;
     note # of page accesses made;
     perform joins;
     extract result;
end processing 2nd query;
end.
```

Figure 9. Pseudo-code for the simulation of the proposed query processing approach

When the program is run, it first looks into the MANAGER file and based on the information there it creates a relation matrix. The relation matrix identifies each attribute with the appropriate relation. It is implemented with a two-dimensional matrix.

Another structure known as the relation-list is made to hold the set E described in the previous section. This has the list of the attributes that are to be projected from that

relation, and the conditions that are attached to any attribute present in that relation. It is implemented using a linked list. File pointers are used for implementation of projections and selections. New temporary relations that are created are represented as files. Procedures responsible for the projection, selection and join operations use the relation-list and a temporary- file-list as inputs. The projection procedure eliminates those attributes that are not needed to answer the query. The selection procedure ensures that all the predicate relationships specified by the query have been considered. The nested-loop method of join is used to perform the join operations. The relations are assumed to be read sequentially.

CHAPTER V

RESULTS AND CONCLUSIONS

The new approach to query optimization was developed and tested for a random sample of queries. Comparison was made with the 'traditional' method of evaluating i.e. the method without using the proposed approach but in all other aspects the same. Initially the database used for testing the proposed technique was created. This sample database was created very much similar to the sample database available on the QMF (Query Management Facility) running under IBM VM/SP (Virtual Machine/System Product) operating system on the mainframe IBM 3090 machine. It was available from the University Computer Center, Oklahoma State University. QMF is a user interface to access data stored in relational databases such as SQL/DS (SQL/Data System) or IBM DB2 (Database 2). The conclusions inferred from the test runs are discussed below.

Evaluation Criteria

Evaluation of the test is based upon the number of page accesses made for processing the given query. This indirectly reflects on the time complexity also, since the most time consuming operation in the evaluation of a query

is in making the disk accesses. It follows that fewer disk accesses made, leads to better performance.

Results

First several runs were made to identify the problems incurred during the development of the simulation process. Later test runs were made querying different number of attributes. Also the number of common selection attributes were varied so as to show the effect of this parameter on the processing. The test results are shown in Figures 10 through 13. The below tables give the average number of disk accesses made for every set of two queries by the two methods.

TABLE I

COMPARISON OF THE NUMBER OF DISK ACCESSES MADE BETWEEN THE TWO METHODS WITH THE NUMBER OF ATTRIBUTES QUERIED = 3

NUMBER OF COMMON SELECTION ATTRIBUTES	NUMBER OF PAGE ACCESSES	
	TRADITIONAL	NEW METHOD
0	60	60
1	60	40
2	60	33
3	60	30

TABLE II

COMPARISON OF THE NUMBER OF DISK ACCESSES MADE BETWEEN THE TWO METHODS WITH THE NUMBER OF ATTRIBUTES QUERIED = 4

NUMBER OF COMMON SELECTION ATTRIBUTES	NUMBER OF PAGE ACCESSES	
	TRADITIONAL	NEW METHOD
0	80	80
1	80	70
2	80	57
3	80	50
4	80	40

TABLE III

COMPARISON OF THE NUMBER OF DISK ACCESSES MADE BETWEEN THE TWO METHODS WITH THE NUMBER OF ATTRIBUTES QUERIED = 5

NUMBER OF COMMON SELECTION ATTRIBUTES	NUMBER OF PAGE ACCESSES	
	TRADITIONAL	NEW METHOD
0	100	100
1	100	80
2	100	73
3	100	70
4	100	60
5	100	50

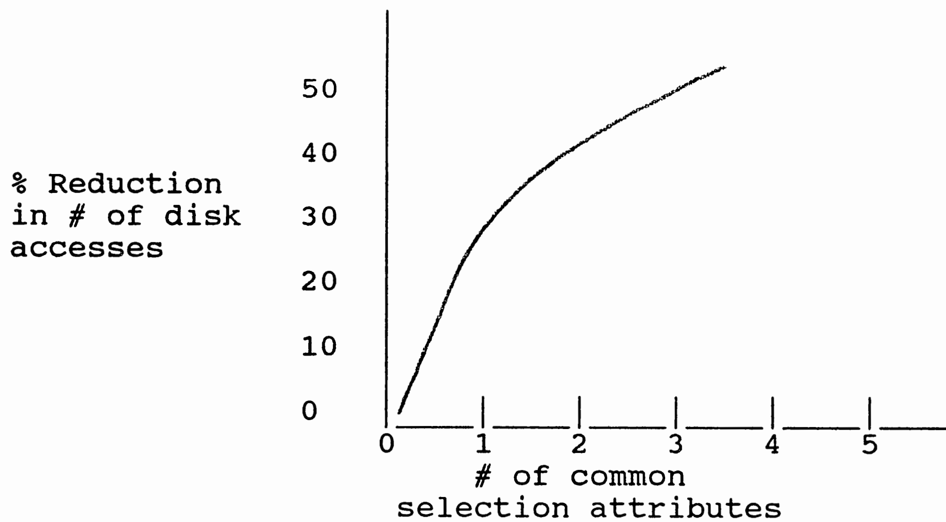

Figure 10. Graph showing the relation between the # of common selection attributes and the percentage reduction in disk accesses when # of attributes queried = 3

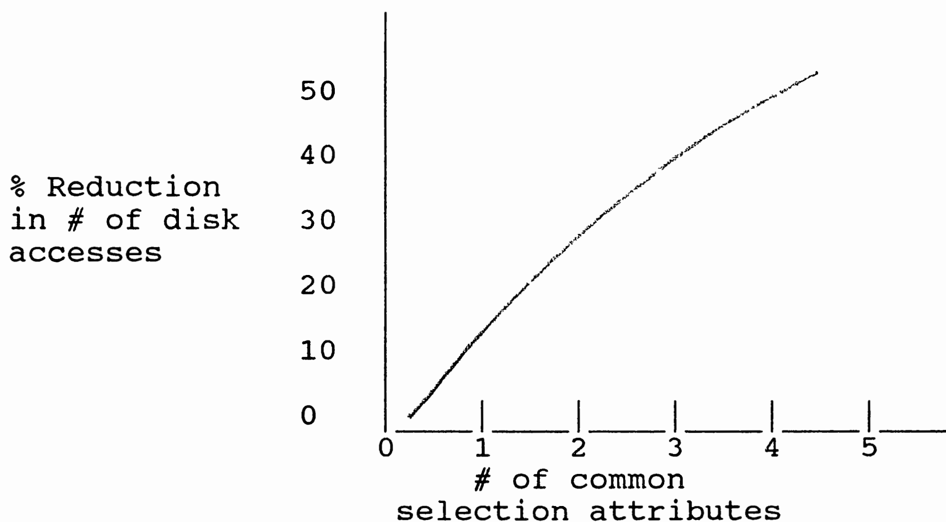

Figure 11. Graph showing the relation between the # of common selection attributes and the percentage reduction in disk accesses when # of attributes queried = 4

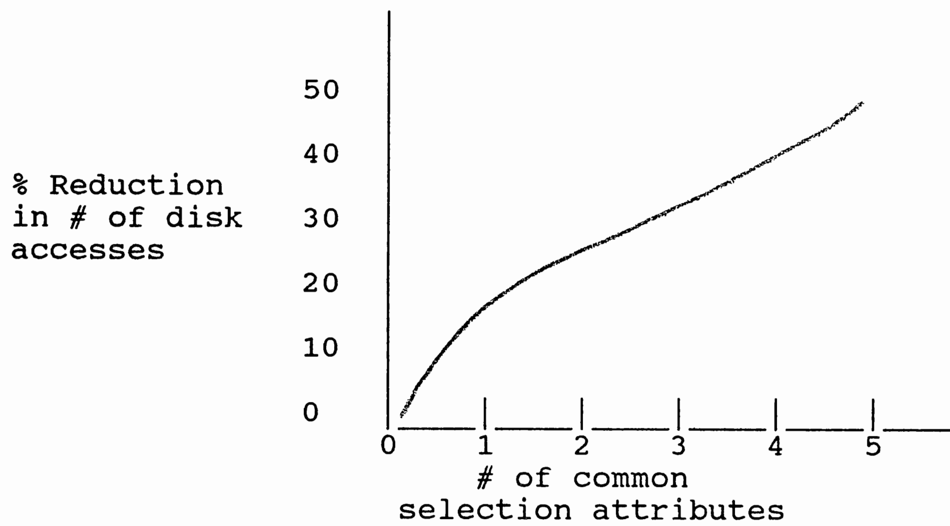

Figure 12. Graph showing the relation between the # of common selection attributes and the percentage reduction in disk accesses when # of attributes queried = 5

Conclusions

The new technique of query optimization was developed with the intention that it would enhance the processing of existing query optimization algorithms. The proposed technique can be used in combination with other algorithms. Based on several test runs made with the simulation program, the following conclusions can be inferred:

i. The proposed method does decrease the number of page accesses made for processing any set of two queries.

ii. On the average, the number of page accesses made (by the proposed method) reduce approximately linearly, with the number of selection attributes that are common between the two queries.

iii. Also a reduction in the size of the temporary tables created is observed, but however the processing time for the queries does increase slightly.

Future Work

The present work was aimed at developing a new technique for query processing in relational databases, based on reading two queries at a time. As a sequence to this work, multi query processing (with more than two) can be attempted. This would be quite useful especially in statistical databases.

Also another possible future direction of this research work is to improve on the software implementation of the proposed technique. A number of refinements and extensions can be suggested. The impact of a system's overhead and the way it affects the overall cost could be studied. The proposed technique is active only when there are some common selection attributes between the two queries. But an improvement can be got even if there is just some relation in common in between the two queries (and not only attributes). There is also scope for future work in this direction.

REFERENCES

[Aho79]
Aho, A.V., Beeri, C., and Ullman, J.D., The Theory of Joins in Relational Databases, ACM Transactions on Database Systems, Vol. 4, No. 3, Sept 89, pp. 297-314.

[Aper88]
Apers, M.G., Data Allocation in Distributed Database Systems, ACM Transactions on Database Systems, Vol. 13, No.3, Sept 88, pp. 263-304.

[Bez86]
Bezalel, G., and Segev, A., Set Query Optimization in Distributed Database Systems, ACM Transactions on Database Systems, Vol. 11, Sept 86, pp 265-293.

[Cha90]
Chakravarthy, U.S., Grant, J., and Minker, J., Logic-Based Approach to Semantic Query Optimization, ACM Transactions on Database Systems, Vol. 15, No. 2, June 90, pp. 162-207.

[Codd70]
Codd, E.F., A Relational Model of Data for Large Shared Data Banks, Communications of the ACM, Vol. 13, No. 6, June 1970.

[Date87]
Date, C.J., An Introduction to Database Systems, Addison-Wesley Publishing Company, Inc. , Reading, MA, 1987.

[Deen85]
Deen, S.M., Amin, R.R., Ofori-Dwumfuo, G.O., and Taylor M.C., The Architecture of a Generalized Distributed Database System - PRECI*, Computer Journal, Vol. 28, No. 3, 1985 pp. 282-290.

[Egy88]
Egyhazy, C., and Triantis, K., A Query Processing Algorithm for Distributed Relational Database Systems, Computer Journal, Vol. 31, No. 1, Feb 88, pp. 34-40.

[Fink88]
Finkelstein, R., and Pascal, F., SQL Database Management Systems, Byte, Vol. 13, Jan 88, pp. 111-114.

[Gall84]
Gallaire, H., Minker, J., and Nicolas, J.M., Logic and Databases: a Deductive Approach, ACM Computing Surveys, Vol. 16, 1984, pp. 153-185.

[Gar89]
Gardy, D., and Puech, C., On the Effect of Join Operations on Relation Sizes, ACM Transactions on Database Systems, Vol. 14, No. 4, Dec 89, pp. 574-603.

[Hev83]
Hevner, A.R., and Yao, S.B., Query Processing on a Distributed Database , IEEE Transactions of Software Engineering, Vol. 9, No. 1, Jan 83, pp. 57-68.

[IBM90]
Query Management Facility, Learners Guide Version 3 Release 1, copyright IBM, 1983, 1990, pp. 204-209.

[Jar84]
Jarke, M., and Koch, J., Query Optimization in Database Systems, Computing Surveys, Vol. 16, No. 2, June 84, pp. 111-152.

[Jar85]
Jarke, M., and Vassiliou, Y., A Framework for Choosing a Database Query Language, ACM Computing Surveys, Vol. 17, No. 3, Sept 85, pp. 313-40.

[Ker82]
Kerschberg, L., Ting P.D., and Yao S.B., Query Optimization in Star Computer Networks, ACM Transactions on Database Systems, Vol. 7, No. 4, Dec 82, pp. 678-711.

[Moh86]
Mohan, C., Lindsay B., and Obermarck R.,Transaction Management in the R* Distributed Database Management System, ACM Transactions on Database Systems, Vol. 11, No. 4, Dec 86, pp. 378-396.

[Ozs89]
Ozsoyoglu, G., Maltos, V., Ozsoyoglu, Z.M., Query Processing Techniques in the Summary-Table-by-Example Database Query Language, ACM Transactions on Database Systems, Vol. 14, Dec 89, pp. 526-573.

[Pal87]
Palley, M.A., and Simonoff, J.S., The Use of Regression Methodology for the Compromise of Confidential Information in Statistical Databases, ACM Transactions on Database Systems, Vol. 12, No. 4, December 87, pp. 593-608.

[QMF90]
Query Management Facility General Information, Version 3 Release 1, IBM Corp., 1990.

[Rob88]
Robie, Jonathan, Fast Data Access, Byte, Vol. 13, Jan 88, pp. 243-250.

[Sel91]
Sellis, T.K., and Shapiro, L., Query Optimization for Nontraditional Database Applications, IEEE Transactions on Software Engineering, Vol. 17, No. 1, Jan 91, pp. 77-86.

[Silb91]
Silberschatz, A., Stonebraker, M., and Ullman, J., Database Systems: Achievements and Opportunities, Communications of the ACM, Vol. 34, No. 10, Oct 91, pp. 110-120.

[Ston89]
Stonebraker, M., The Distributed Database Decade, Datamation, Vol. 35, Sept 15 '89, pp. 38-39.

[Ston91]
Stonebraker, M., and Kemnitz, G., The POSTGRES Next Generation Database Management System, Communications of the ACM, Vol. 34, No. 10, Oct 91, pp. 78-92.

[Vet89]
De Vet, John H.M., A Practical Algorithm for evaluating Database Queries, Software-Practice and Experience, Vol. 19, May 85, pp. 491-504.

[Wald90]
Wald, J.A., and Sorenson, P.G., Explaining Ambiguity in a Formal Query Language, ACM Transactions on Database Systems, Vol. 15, No. 2, June 90, pp. 125-161.

[Wha90]
Whang, K., and Krishnamurthy, R., Query Optimization in a Memory-Resident Domain Relational Calculus Database System, ACM Transactions on Database Systems, Vol. 15, No. 1, March 90, pp. 67-95.

[Wong76]
Wong, E., and Youssefi, K., Decomposition - A Strategy for Query Processing, ACM Transactions on Database Systems, Vol. 1, No. 3, Sept 76, pp. 223-241.

[Yu84]
Yu, C.T., and Chang, C.C., Distributed Query Processing, Computing Surveys, Vol. 16, No. 4, Dec 84, pp. 399-408.

APPENDIX

PROGRAM IMPLEMENTATION

PROGRAM FOR THE SIMULATION OF THE PROPOSED QUERY PROCESSING TECHNIQUE

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>


#define  MAX_NUM_RELATIONS       7
#define  MAX_REL_NAME_LEN       10
#define  MAX_NUM_ATTRS_IN_REL   15
#define  MAX_ATTR_NAME_LEN      10
#define  MAX_NUM_ATTRS_QRYD     15
#define  MAX_NUM_CONDNS         20
#define  MAX_NUM_TPLS_PER_REL  150
#define  MAX_TPL_LEN           150


/*********************************
*                               *
*        GLOBAL VARIABLES       *
*                               *
*********************************/


typedef  char      Atr_name [MAX_ATTR_NAME_LEN];
typedef  Atr_name  Q_ary [MAX_NUM_ATTRS_QRYD];

typedef  struct    {
         char      rel_name [MAX_REL_NAME_LEN];
         Atr_name  attr_ary [MAX_NUM_ATTRS_IN_REL];
         Atr_name  key_ary  [MAX_NUM_ATTRS_IN_REL];
         int       tpl_sz;
         int       num_tp;
         } Mtrx_nd;

typedef  struct    {
         Atr_name  lhs;
         Atr_name  mid;
         Atr_name  rhs;
         } Condn;
```

```
typedef  struct     {
         Atr_name   lhs;
         Atr_name   mid1;
         Atr_name   rhs1;
         Atr_name   mid2;
         Atr_name   rhs2;
         } Common;

typedef  struct P_node   {
         struct P_node   *next;
         Atr_name        data;
         } Proj_nd, *P_nd_ptr;

typedef  struct C_node   {
         struct C_node   *next;
         Condn           data;
         } Condn_nd, *C_nd_ptr;

typedef  struct L_node   {
         char            rel_name [MAX_REL_NAME_LEN];
         P_nd_ptr        p_ptr;
         C_nd_ptr        c_ptr;
         struct L_node   *next;
         } List_nd, *List_nd_ptr;

typedef  struct T_node   {
         FILE            *fp;
         char            rel_name [MAX_REL_NAME_LEN];
         Atr_name        atr [MAX_NUM_ATTRS_IN_REL];
         int             tpl_sz;
         int             num_tp;
         struct T_node   *next;
         } Temp_nd, *Temp_nd_ptr;

typedef  Mtrx_nd   Matrix  [MAX_NUM_RELATIONS];
typedef  Condn     C_ary   [MAX_NUM_CONDNS];
typedef  Common    Cmn_ary [MAX_NUM_CONDNS];

int      num_pg_accs,QRY_NUM=1;
Matrix   matrix;
Q_ary    q_ary1,q_ary2;
C_ary    c_ary1,c_ary2;
Cmn_ary  cmn_ary;

List_nd_ptr  head;
Temp_nd_ptr  t_hd = NULL;
```

```
/********************************
*                               *
*          MAIN PROGRAM         *
*                               *
*********************************

     This is the main driver routine. It calls the other
functions (twice if necessary).
No input parameters for this routine.
*/


main ()

{
        int     PRSNT;
        char    ans[10];

        printf ("\n\n");
        create_rel_matrix (matrix);
        read_query (q_ary1,c_ary1);
        printf ("\nDo you want to enter 2nd query?");
        printf (" (Answer Y or N) : ");
        ans[0] = 'N';
        scanf ("%s",ans);
        if (ans[0] == 'Y') {
                read_query (q_ary2,c_ary2);
                PRSNT = check_sel_attrs (c_ary1,c_ary2);
                if (PRSNT)
                        process_sel_attrs ();
        }
        make_rel_list (&head,q_ary1,c_ary1);
        make_sels_projs ();
        perform_joins ();
        print_results (q_ary1);
        printf ("\n\n");

        if (ans[0] == 'Y') {
                t_hd = NULL;
                QRY_NUM = 2;
                make_rel_list (&head,q_ary2,c_ary2);
                make_sels_projs ();
                perform_joins ();
                print_results (q_ary2);
                printf ("\n\n");
        }
        printf ("\nTotal # of page accesses =
%d\n",num_pg_accs);
}
```

```
/*********************************
*                               *
*        CREATE_REL_MATRIX      *
*                               *
*********************************

        This function is to read the existing database files
and create a matrix to relate each attribute to its
respective file.
Input:
mtrx -    the relation matrix containing information about
          different relations present in the database.
*/

create_rel_matrix (mtrx)
Matrix  mtrx;

{
        int     i=0,k=0,DONE=0,
                num_rels, f_pos;
        char    *c,s[81],ch, str [MAX_REL_NAME_LEN];
        FILE    *f1, *f2;

        f1 = fopen ("MNGR","r");
        fscanf (f1,"%s",str);
        while  (!DONE)  {
                strcpy (mtrx[i].rel_name,str);
                f2 = fopen (str,"r");

                fgets (s,80,f2);
                k = strlen (s);
                s[k-1] = '\0';
                c = strtok (s," ");
                k = 0;
                strcpy (mtrx[i].attr_ary[k++],c);
                while (*c)  {
                        c = strtok ('\0'," ");
                        strcpy (mtrx[i].attr_ary[k++],c);
                }
                strcpy (mtrx[i].attr_ary[k-1], "^");

                f_pos = ftell (f2);
                mtrx[i].tpl_sz = f_pos;
                fclose (f2);

                k = 0;
                fscanf (f1,"%s",str);
                fscanf (f1,"%d",&mtrx[i].num_tp);
                fscanf (f1,"%s",str);
                while ((strcmp (str,".") != 0)  &&
                       (strcmp (str,"^") != 0))  {
                        strcpy (mtrx[i].key_ary[k++],str);
                        fscanf (f1,"%s",str);
                }
```

```
                strcpy (mtrx[i].key_ary[k], "^");

                if (strcmp (str,"^") == 0)
                        DONE = 1;
                else
                        fscanf (f1,"%s",str);
                i++;
        }
        strcpy (mtrx[i].rel_name,"^");
        num_rels  = i;
}
```

```
/*********************************
*                               *
*            READ_QUERY         *
*                               *
*********************************

        This function reads the input query and identifies
those attributes that are needed to be displayed in the
final result.
Input:
q_ary -   array to contain attributes needed in the result
c_ary -   array to contain conditions read alongwith
*/


read_query (q_ary,c_ary)
Q_ary   q_ary;
C_ary   c_ary;

{
        int      i=0, j=0, DONE=0,
                 num_condns=0, num_res_attrs=0;
        Atr_name str;

        printf ("\nPlease type in query: \n");
        scanf ("%s",str);
        j = strlen (str);
        while  (!DONE)  {
                if (str[j-1] == ';')  {
                        str[j-1] = '\0';
                        DONE = 1;
                }
                if (str[j-1] == ',')
                        str[j-1] = '\0';
                strcpy (q_ary[i++],str);
                if (!DONE)  {
```

```
                        scanf ("%s",str);
                        j = strlen (str);
                        if (strcmp (str,";") == 0)
                                DONE = 1;
                }
        }
        strcpy (q_ary[i],"^");
        num_res_attrs = i;

        i = 0;
        DONE = 0;
        scanf ("%s",str);
        if (strcmp (str,".") == 0)
                DONE = 1;
        while (!DONE)  {
                strcpy (c_ary[i].lhs,str);
                scanf ("%s",str);
                strcpy (c_ary[i].mid,str);
                scanf ("%s",str);
                j = strlen (str);
                if (str[j-1] == ',')
                        str[j-1] = '\0';
                if (str[j-1] == '.')  {
                        str[j-1] = '\0';
                        DONE = 1;
                }
                strcpy (c_ary[i++].rhs,str);
                if (!DONE)  {
                        scanf ("%s",str);
                        if (strcmp (str, ".") == 0)
                                DONE = 1;
                }
        }

        strcpy (c_ary[i].lhs, "^");
        num_condns = i;

}
```

```
/********************************
*                                *
*          CHECK_SEL_ATTRS       *
*                                *
********************************
```

This functions is for getting the attributes common to the two relations that are passed to it as parameters. It stores the common attributes in the array c_atr_ary.
Input:
a1 - array containing the conditions of query 1
a2 - array containing the conditions of query 2
*/

```
check_sel_attrs (a1,a2)
C_ary      a1,a2;

{
        int  i1=0,i2=0,k1,k2,
             j=0,PRSNT=0;

        while  (strcmp (a1[i1++].lhs,"^") != 0);
        while  (strcmp (a2[i2++].lhs,"^") != 0);
        for (k1 = 0; k1 < (i1-1); k1++)
                for (k2=0; k2 < (i2-1); k2++)
                        if (strcmp (a1[k1].lhs, a2[k2].lhs)
                              == 0)  {
                                strcpy (cmn_ary[j].lhs,
                                        a1[k1].lhs);
                                strcpy (cmn_ary[j].mid1,
                                        a1[k1].mid);
                                strcpy (cmn_ary[j].rhs1,
                                        a1[k1].rhs);

                                strcpy (cmn_ary[j].mid2,
                                        a2[k2].mid);
                                strcpy (cmn_ary[j].rhs2,
                                        a2[k2].rhs);
                                j++;
                        }
        strcpy (cmn_ary[j].lhs,"^");
        if (j != 0)
                PRSNT = 1;
        return (PRSNT);
}
```

```
/*********************************
*                               *
*        PROCESS_SEL_ATTRS      *
*                               *
*********************************

        This procedure is for processing the selection
attributes only that which are common to both the queries.
It modifies the relation matrix as necessary.
*/

process_sel_attrs ()

{
        int     i,k,tpl_sz,num_tp,mtrx_loc,TRUE;
        long    j1,j2;
        char    r_name[10],val_rd[10],line [MAX_TPL_LEN];
        FILE    *f1,*f2;


        for (i=0; strcmp(cmn_ary[i].lhs,"^") != 0; i++)  {
                find_rel (cmn_ary[i].lhs,r_name,&mtrx_loc);
                tpl_sz = matrix[mtrx_loc].tpl_sz;
                num_tp = matrix[mtrx_loc].num_tp;
                f1 = fopen (r_name,"r");
                strcat (r_name,"N");
                f2 = fopen (r_name,"w");
                fgets   (line,MAX_TPL_LEN,f1);
                fprintf (f2,"%s",line);
                strcpy (matrix[mtrx_loc].rel_name,r_name);

                fseek  (f1,0,0);
                strcpy (val_rd,"^");
                while (strcmp(val_rd,cmn_ary[i].lhs) != 0)
                        fscanf (f1,"%s",val_rd);
                j1 = ftell (f1) - strlen (val_rd);

                for (k=1; k<=num_tp; k++)  {
                        num_pg_accs++;
                        j2 = (k*tpl_sz) + j1;
                        fseek  (f1,j2,0);
                        fscanf (f1,"%s",val_rd);
                        TRUE = compare
(val_rd,cmn_ary[i].rhs1,
                                        cmn_ary[i].mid1);
                        if (!TRUE)
                                TRUE = compare
(val_rd,cmn_ary[i].rhs2,

cmn_ary[i].mid2);
                        if (TRUE)  {
                                j2 = k * tpl_sz;
                                fseek   (f1,j2,0);
```

```
                                        fgets
(line,MAX_TPL_LEN,f1);
                                        fprintf (f2,"%s",line);
                                }
                        }
                        j2 = ftell (f2);
                        matrix[mtrx_loc].num_tp = j2/tpl_sz - 1;
                        fclose (f1);
                        fclose (f2);
                }
}
```

```
/*********************************
*                               *
*          MAKE_REL_LIST        *
*                               *
*********************************
```

This function is for making the list of files that are to be read for processing the given query and attaching the necessary conditions to that list.

```
Input:
hd -      head of the list containing the temp relations
          created
q_ary -   array containing the result attributes of query
c_ary -   array containing the restrictions of the query
*/

make_rel_list (hd,q_ary,c_ary)
List_nd_ptr     *hd;
Q_ary           q_ary;
C_ary           c_ary;

{
        int             i=0,FOUND,mtrx_loc;
        char            r_name [MAX_REL_NAME_LEN];
        List_nd_ptr     p1,p2;

        *hd = NULL;
        for (i=0; (strcmp (q_ary[i],"^") != 0); i++)  {
                find_rel  (q_ary[i],r_name,&mtrx_loc);
                if ((*hd) == NULL)  {
                        *hd = (List_nd_ptr) malloc
                              (sizeof (List_nd));
                        *hd = NULL;
                        make_new_node (hd,r_name,q_ary[i],
                              matrix[mtrx_loc].key_ary);
                }
```

```
                else  {
                        locate_rel (r_name, (*hd),
                                    &p1,&FOUND);
                        if (FOUND)
                                add_attr (p1,q_ary[i]);
                        else
                          make_new_node (&p1, r_name,
                         q_ary[i],matrix[mtrx_loc].key_ary);
                }
        }

        for (i=0; strcmp (c_ary[i].lhs,"^") != 0; i++)  {
                find_rel (c_ary[i].lhs,r_name,&mtrx_loc);
                locate_rel (r_name,(*hd),&p1,&FOUND);
                if (FOUND)
                        add_condn (p1,c_ary[i]);
                else  {
                        make_new_node (&p1,r_name,  matrix
                                      [mtrx_loc].key_ary[0],
                                  matrix[mtrx_loc].key_ary);
                        p2 = p1->next;
                        add_condn (p2,c_ary[i]);
                }
        }
}
```

```
/*********************************
*                                *
*             FIND_REL           *
*                                *
**********************************

        This function is to locate the name of the relation
in which a particular attribute is found. It returns the
name string.
Input:
an -     name of the attribute to be matched to its
         corresponding relation
rn -     relation name that is matched with an
loc -    integer to show location of rn in the matrix
*/

find_rel (an,rn,loc)
Atr_name an;
char     rn[];
int      *loc;

{
```

```
        int     i=0,k, DONE=0, FOUND,OVER;

        while (!DONE)  {
                FOUND = 0;
                OVER  = 0;
                k = 0;
                while ((!FOUND) && (!OVER))
                  if (strcmp (matrix[i].attr_ary[k],
                              an) == 0)
                     FOUND = 1;
                  else
                    if (strcmp (matrix[i].attr_ary[k],
                                "^") == 0)
                                       OVER = 1;
                    else
                                        k++;
                        if (OVER)
                                i++;
                        if (FOUND)  {
                                DONE = 1;
                                strcpy (rn,
                                        matrix[i].rel_name);
                                *loc = i;
                        }
                        if (strcmp (matrix[i].rel_name,
                                    "^") == 0)  {
                                DONE = 1;
                                strcpy (rn, "not found");
                        }
        }
}
```

```
/*********************************
*                               *
*         MAKE_NEW_NODE         *
*                               *
*********************************
```

This function is for making a new relation node in the list and attaching it to the end. It also modifies the needed pointers.

Input:

p1 - ptr to show location in list (holding involved relation names) where new node is to be inserted*/

rn - relation name which is to be included in the list

attr - attribute in that relation which is to be projected

k_ary - array to hold key attributes of the relation

```
*/

make_new_node (p1,rn,attr,k_ary)
List_nd_ptr    *p1;
char           rn[],attr[];
Atr_name       k_ary[];

{
        int            i,k;
        List_nd_ptr  n1;
        P_nd_ptr     t1,t2;

        n1 = (List_nd_ptr) malloc (sizeof (List_nd));
        n1->p_ptr = (P_nd_ptr) malloc (sizeof (Proj_nd));
        n1->c_ptr = (C_nd_ptr) malloc (sizeof (Condn_nd));
        strcpy (n1->rel_name,rn);
        n1->c_ptr = NULL;
        n1->next  = NULL;

        /* copy key atributes first */
        strcpy (n1->p_ptr->data, k_ary[0]);
        n1->p_ptr->next  = NULL;
        t1 = n1->p_ptr;
        for (i=1; (strcmp (k_ary[i], "^") != 0); i++)  {
                t1->next = (P_nd_ptr) malloc (sizeof
                           (Proj_nd));
                strcpy (t1->next->data, k_ary[i]);
                t1 = t1->next;
        }
        t1->next = NULL;
        add_attr (n1,attr);

        if ((*p1) == NULL)
                (*p1) = n1;
        else
                (*p1)->next = n1;
}
```

```
/**********************************
*                                 *
*          LOCATE_REL             *
*                                 *
**********************************
```

This function is for checking the relation list to see whether the given relation is already in it or not.
Input:
rn - relation whose presence in the list is to be found

```
hd -       head of the list containing involved relations
ptr -      ptr returns location of rel in the list
FOUND -    boolean variable to indicate success in search
*/

locate_rel (rn,hd,ptr,FOUND)
char          rn [];
List_nd_ptr   hd,*ptr;
int           *FOUND;

{
        int           DONE;
        List_nd_ptr   t1;

        *FOUND = 0; DONE = 0;
        *ptr = hd;
        while ((!(*FOUND))  && (!DONE))  {
                if (strcmp ((*ptr)->rel_name, rn) == 0)
                        *FOUND = 1;
                else {
                        t1 = *ptr;
                        *ptr = (*ptr)->next;
                }
                if (*ptr == NULL) {
                        DONE = 1;
                        *ptr = t1;
                }
        }
}
```

```
/*********************************
*                               *
*             ADD_ATTR          *
*                               *
*********************************
```

This function is for adding a given attribute to the proper relation in the list of relations.

```
Input:
p1 -       ptr to indicate location of the relation in the
           list
attr -     attribute needed in the result (is added)
*/

add_attr (p1,attr)
List_nd_ptr   p1;
char          attr[];
```

```
{
        int       ALRDY_PRSNT=0;
        P_nd_ptr  t1,t2;

        t1 = p1->p_ptr;
        while ((!ALRDY_PRSNT)  &&  (t1 != NULL))
                if (strcmp (t1->data,attr) == 0)
                        ALRDY_PRSNT = 1;
                else  {
                        t2 = t1;
                        t1 = t1->next;
                }
        if (!ALRDY_PRSNT)  {
                t2->next = (P_nd_ptr) malloc (sizeof
                           (Proj_nd));
                strcpy (t2->next->data,attr);
                t2->next->next = NULL;
        }
}
```

```
/*********************************
*                                *
*           ADD_CONDN            *
*                                *
**********************************

        This function is for adding a given selection
criteria to the proper relation in the list of relations.
Input:
p1 -     ptr to indicate relation location in list
condn -  condition to be added to that relation entry
*/

add_condn (p1,condn)
List_nd_ptr   p1;
Condn         condn;

{
        int       DONE=0;
        C_nd_ptr  t1,t2;

        t1 = p1->c_ptr;
        t2 = t1;
        while (!DONE)  {
                if (t1 == NULL)
                        DONE = 1;
                else  {
                        t2 = t1;
```

```
                        t1 = t1->next;
                }
        }
        if (t2 == NULL)  {
                p1->c_ptr = (C_nd_ptr) malloc (sizeof
                                (Condn_nd));
                p1->c_ptr->data  = condn;
                p1->c_ptr->next = NULL;
        }
        else  {
                t2->next = (C_nd_ptr) malloc (sizeof
                              (Condn_nd));
                t2->next->data  = condn;
                t2->next->next = NULL;
        }       /* end main else */
}




/**********************************
*                                *
*         MAKE_SELS_PROJS        *
*                                *
**********************************

        This function is for making the actual selections on
the relation tables and then projecting the required
attributes to form new relations.
*/

make_sels_projs ()

{
        int             i,j,row_num,l,
                        tpl_sz,num_tp,DISK,
                        row [MAX_NUM_TPLS_PER_REL];
        Atr_name        val;
        List_nd_ptr     t1;
        P_nd_ptr        t2;
        C_nd_ptr        t3;
        FILE            *fpp [MAX_NUM_ATTRS_IN_REL],
                        *fpc [MAX_NUM_ATTRS_IN_REL];

        t1 = head;
        while (t1 != NULL)  {
                DISK = 0;
                if (t1->rel_name[strlen(t1->rel_name)-1]
                    != 'N')
                        DISK = 1;
```

```
                get_rel_parms (t1->rel_name,
                               &tpl_sz,&num_tp);
                build_file_ptrs1 (t1,fpp,fpc,tpl_sz);
                for (row_num=0; row_num < num_tp; row_num++)
                {
                        t3 = t1->c_ptr;
                        make_selections  (t3,fpc,tpl_sz,
                                          row,row_num);
                        make_projections (t1,fpp,tpl_sz,
                                          row,row_num);
                        if (DISK)
                                num_pg_accs++;
                }

                check_temp_rel (t1,&t_hd,fpp,fpc);
                t1 = t1->next;
        }
}
```

```
/*********************************
*                               *
*          GET_REL_PARMS        *
*                               *
*********************************

        This function is for getting the size of each tuple
of the given relation and also the num of tuples in it, by
referring to the matrix.
Input:
rel -     relation whose parameters are needed
sz -      size of one tuple in the relation
num -     number of tuples in the relation
*/

get_rel_parms (rel,sz,num)
char  rel[];
int   *sz,*num;

{
        int   i=0,DONE=0;

        while (!DONE)  {
                if (strcmp(rel,matrix[i].rel_name) == 0)
                        DONE = 1;
                else
                        i++;
        }
```

```
        *sz  = matrix[i].tpl_sz;
        *num = matrix[i].num_tp;
}
```

```
/*********************************
*                                *
*        BUILD_FILE_PTRS1        *
*                                *
*********************************

        This function is for building the file pointers to
the given relations so as to read each attribute value.
Input:
t1 -      ptr to indicate relation position in the list
fpp -     array containing file ptrs pointing to result
          attributes in the relation
fpc -     array containing file ptrs pointing to condition
          attributes in the relation
tpl_sz -  size of each tuple in the relation
*/

build_file_ptrs1 (t1,fpp,fpc,tpl_sz)
List_nd_ptr  t1;
FILE         *fpp[MAX_NUM_ATTRS_IN_REL],
             *fpc[MAX_NUM_ATTRS_IN_REL];
int          tpl_sz;

{
        int       i,j,k;
        Atr_name  attr,lhs,str;
        P_nd_ptr  t2;
        C_nd_ptr  t3;
        FILE      *fopen ();


        t2 = t1->p_ptr;
        for (i=0; t2 != NULL; i++)  {
                fpp[i] = fopen (t1->rel_name,"r");
                strcpy (attr,"^");
                while (strcmp(attr,t2->data) != 0)
                        fscanf (fpp[i],"%s",attr);
                j = ftell (fpp[i]) - strlen(attr);
                fseek (fpp[i],j,0);

                t2 = t2->next;
        }

        t3 = t1->c_ptr;
```

```
        for (i=0; t3 != NULL; i++)  {
                fpc[i] = fopen (t1->rel_name,"r");
                strcpy (lhs,"^");
                while (strcmp(lhs,t3->data.lhs) != 0)
                        fscanf (fpc[i],"%s",lhs);
                j = ftell (fpc[i]) - strlen(lhs) + tpl_sz;
                fseek (fpc[i],j,0);

                t3 = t3->next;
        }
}




/**********************************
*                                *
*         MAKE_SELECTIONS        *
*                                *
**********************************

        This function is for making the required selections
in the given relations.
Input:
p1 -      ptr to show position of conditions in the query
          pertaining to this relation
fpc -     array containing file ptrs pointing to the
          condition attributes in the relation
tpl_sz -  size of each tuple in the relation
row -     array containing numbers of those tuples that are
          selected
row_num - number of the selected row
*/

make_selections (p1,fpc,tpl_sz,row,row_num)
C_nd_ptr  p1;
FILE      *fpc[];
int       tpl_sz, row[],row_num;

{
        int       i=0,j,k,TRUE,ONCE_FL=0,r2[6];
        Atr_name  val_rd;
        C_nd_ptr  p2,p3;

        row [row_num] = 0;
        while (p1 != NULL)  {
                fscanf (fpc[i],"%s",val_rd);
                j = ftell (fpc[i]) -
                    strlen (val_rd) + tpl_sz;
                TRUE = compare (val_rd,p1->data.rhs,
                                p1->data.mid);
                if (!TRUE)
```

```
                    ONCE_FL=1;
                fseek (fpc[i], j,0);

                p1 = p1->next;
                i++;
        }
        if (!ONCE_FL)
                row[row_num] = 1;
}
/*********************************
*                               *
*             COMPARE           *
*                               *
*********************************

        This function is for comparing the value present in
the relation  with the expected value, and it returns an
indication to the comparision performed.
Input:
val_rd -     value read from the relation
val_exptd - value asked for in the query
mid -        the operand connecting the above two values
*/

compare  (val_rd,val_exptd,mid)
Atr_name  val_rd,val_exptd,mid;

{
        int     k,TRUE=0;
        char    c;

        switch (mid[0])  {
        case '='  :  if (strcmp(val_rd,val_exptd) == 0)
                             TRUE = 1;
                     break;
        case '!'  :  if (strcmp(val_rd,val_exptd) != 0)
                             TRUE = 1;
                     break;
        case '>'  :  k = strlen (mid);
                     if (k == 1)  {
                             if (strcmp(val_rd,
                                        val_exptd) > 0)
                                     TRUE = 1;
                     }
                     else
                             if ((strcmp(val_rd,
                                        val_exptd) >  0)  ||
                                 (strcmp(val_rd,
                                         val_exptd) == 0))
                                     TRUE = 1;
                     break;
        case '<'  :  k = strlen (mid);
                     if (k == 1)  {
```

```
                                if (strcmp(val_rd,
                                           val_exptd) < 0)
                                        TRUE = 1;
                        }
                        else
                                if ((strcmp(val_rd,
                                           val_exptd) <  0)  ||
                                    (strcmp(val_rd,
                                            val_exptd) == 0))
                                        TRUE = 1;
                        break;
        default    :    break;
        }

        return (TRUE);
}
```

```
/*********************************
*                               *
*       MAKE_PROJECTIONS        *
*                               *
*********************************

        This routine is for projecting the needed attributes
to a temporary file from where the joins can take place.
Input:
t1 -      ptr indicating relation position in the list
fpp -     array containing file ptrs pointing to the
          attributes that are to be projected
tpl_sz -  size of each tuple in the relation
row -     array containing the selected rows
row_num - number of the selected row
*/

make_projections (t1,fpp,tpl_sz,row,row_num)
List_nd_ptr  t1;
FILE         *fpp[];
int          tpl_sz,row[], row_num;

{
        int             i=0,j;
        Atr_name        val_rd;
        P_nd_ptr        p2;
        Temp_nd_ptr     tr1;

        if (row_num == 0)
                create_temp_rel (t1,fpp,tpl_sz);
```

```
        if (row [row_num] == 1)  {
                tr1 = t_hd;
                strcat (t1->rel_name,"1");
                while (strcmp (tr1->rel_name,t1->rel_name)
                         != 0)
                        tr1 = tr1->next;
                t1->rel_name [strlen(t1->rel_name)-1] =
                              '\0';
                p2 = t1->p_ptr;
                while (p2 != NULL)  {
                        fscanf  (fpp[i], "%s",val_rd);
                        fprintf (tr1->fp,"%-10s",val_rd);
                        j = ftell(fpp[i]) + tpl_sz -
                            strlen(val_rd);
                        fseek (fpp[i++],j,0);
                        p2 = p2->next;
                }
                fprintf (tr1->fp,"\n");
        }
        else  {
                p2 = t1->p_ptr;
                while (p2 != NULL)  {
                        j = ftell(fpp[i]) + tpl_sz;
                        fseek (fpp[i++],j,0);
                        p2 = p2->next;

                }
        }
}




/**********************************
*                                *
*        CREATE_TEMP_REL         *
*                                *
**********************************

        This function creates a temporary file to have the
result of the selections and projections made on a relation.
Input:
t1 -      ptr indicating relation position in the list
fpp -     array containing file ptrs pointing to those
          attributes that are to be projected
tpl_sz -  size of each tuple in the relation
*/

create_temp_rel (t1,fpp,tpl_sz)
List_nd_ptr  t1;
FILE         *fpp[];
```

```
int            tpl_sz;

{
        int          i=0,k=0,j;
        char         t_fl [MAX_REL_NAME_LEN];
        P_nd_ptr     p2;
        Temp_nd_ptr n1,tr1,tr2;

        strcpy (t_fl,t1->rel_name);
        strcat (t_fl,"1");
        n1 = (Temp_nd_ptr) malloc (sizeof (Temp_nd));
        strcpy (n1->rel_name,t_fl);
        n1->fp   = fopen (t_fl,"w");
        n1->next = NULL;
        p2 = t1->p_ptr;

        while (p2 != NULL)  {
                fscanf  (fpp[i],"%s",n1->atr[k]);
                fprintf (n1->fp,"%-10s",n1->atr[k]);
                j = ftell (fpp[i]) - strlen(n1->atr[k]) +
                    tpl_sz;
                fseek (fpp[i],j,0);
                i++; k++;
                p2 = p2->next;
        }
        fprintf   (n1->fp,"\n");
        strcpy    (n1->atr[k],"^");
        j = ftell (n1->fp);
        n1->tpl_sz = j;

        if (t_hd == NULL)  {
                t_hd = (Temp_nd_ptr) malloc
                       (sizeof(Temp_nd));
                t_hd = n1;
        }
        else  {
                tr1 = t_hd;
                tr2 = tr1;
                while (tr1 != NULL)  {
                        tr2 = tr1;
                        tr1 = tr1->next;
                }
                tr2->next = (Temp_nd_ptr) malloc
                            (sizeof(Temp_nd));
                tr2->next = n1;
        }
}
```

```
/*********************************
*                               *
*         CHECK_TEMP_REL        *
*                               *
*********************************

        This function checks whether the temporary relation
created has any tuples in it. If not then it removes this
relation from the temp-rel list.
Input:
t1 -      ptr to indicate relation position in the list
t_hd -    head of the list
fpp -     array containing ptrs to the projection attributes
          of the relation
fpc -     array containing ptrs to the selection attributes
          of the relation
*/

check_temp_rel (t1,t_hd,fpp,fpc)
List_nd_ptr      t1;
Temp_nd_ptr      *t_hd;
FILE             *fpp[MAX_NUM_ATTRS_IN_REL],
                 *fpc[MAX_NUM_ATTRS_IN_REL];

{
        int             j;
        Temp_nd_ptr     tr1,tr2;
        P_nd_ptr        t2;
        C_nd_ptr        t3;

        t2 = t1->p_ptr;
        for (j=0; t2 != NULL; j++)   {
                fclose (fpp[j]);
                t2 = t2->next;
        }
        t3 = t1->c_ptr;
        for (j=0; t3 != NULL; j++)   {
                fclose (fpc[j]);
                t3 = t3->next;
        }

        tr1 = *t_hd;
        strcat (t1->rel_name,"1");
        while (strcmp (tr1->rel_name,t1->rel_name) != 0)
                tr1 = tr1->next;
        t1->rel_name [strlen(t1->rel_name)-1] = '\0';
        j = ftell (tr1->fp);
        tr1->num_tp = (j/tr1->tpl_sz) - 1;
        fclose (tr1->fp);

        if (tr1->num_tp < 1)   {
                if (tr1 == *t_hd)
                        *t_hd = (*t_hd)->next;
                else   {
```

```
                        tr2 = *t_hd;
                        strcat (t1->rel_name,"1");
                        while (strcmp (tr2->next->rel_name,
                                       t1->rel_name) != 0)
                                tr2 =tr2->next;
                        t1->rel_name
                         [strlen(t1->rel_name)-1] = '\0';
                        tr2->next = tr2->next->next;
                }
        }
}
```

```
/**********************************
*                                *
*          PERFORM_JOINS         *
*                                *
**********************************

        This function performs the needed joins between the
different relational tables as necessary so as to get a
final relation containing the required attributes.
*/

perform_joins ()

{
        int             DONE=0,PRSNT;
        Atr_name        c_atr_ary [MAX_NUM_ATTRS_IN_REL];
        Temp_nd_ptr     t1,t2;
        char            ch[10];

        t1 = t_hd;
        t2 = t1->next;
        if (t2 == NULL)
                DONE = 1;
        while (!DONE)  {
                PRSNT = get_common_atrs (c_atr_ary,
                                  t1->atr,t2->atr);
                if (PRSNT)  {
                        perform_join2 (c_atr_ary,t1,t2);
                        t1 = t1->next;
                        t2 = t1->next;
                }
                else
                        t2 = t2->next;
                if (t2 == NULL)
                        DONE = 1;
```

```
        }
        t_hd = t1;
}




/********************************
*                               *
*        GET_COMMON_ATRS        *
*                               *
*********************************

        This functions is for getting the attributes common
to the two relations that are passed to it as parameters. It
stores the common attributes in the array c_atr_ary.
Input:
c_atr_ary -    array containing the attributes that are
               common to the two queries*/
a1 -           array containing selection attr.'s of query 1
a2 -           array containing selection attr.'s of query 2
*/

get_common_atrs (c_atr_ary,a1,a2)
Atr_name   c_atr_ary[],a1[],a2[];

{
        int  i1=0,i2=0,k1,k2,
             j=0,PRSNT=0;

        while  (strcmp (a1 [i1++],"^") != 0);
        while  (strcmp (a2 [i2++],"^") != 0);
        for (k1 = 0; k1 < (i1-1); k1++)
                for (k2=0; k2 < (i2-1); k2++)  {
                        if (strcmp (a1[k1], a2[k2]) == 0)  {
                                strcpy (c_atr_ary[j],
                                        a1[k1]);
                                j++;
                        }
                }

        strcpy (c_atr_ary[j],"^");
        if (j != 0)
                PRSNT = 1;
        return (PRSNT);
}
```

```
/*********************************
*                               *
*          PERFORM_JOIN2        *
*                               *
*********************************

        This function performs the actual join between two
given relations.
Input:
ary -  array to contain the key attributes of new rel
t1 -   ptr to indicate relation 1's position in the temp
       relation list
t2 -   ptr to indicate relation 2's position in the temp
       relation list
*/

perform_join2 (ary,t1,t2)
Atr_name     ary[];
Temp_nd_ptr  t1,t2;

{
        int       n1,n2,k,j,hdr_len,
                  ONCE_FL,DONE;
        Atr_name  str,s1,s2;
        FILE      *fp1[MAX_NUM_ATTRS_IN_REL],
                  *fp2[MAX_NUM_ATTRS_IN_REL],*fp;

        strcpy (str,t2->rel_name);
        strcat (str,"2");
        /*
        if (QRY_NUM == 2)
                strcat (str,"T");
        */
        fp = fopen (str,"w");
        create_join_hdr (ary,t1,t2,fp,&hdr_len);

        for (n1=0; n1<t1->num_tp; n1++)  {
                build_file_ptrs2 (ary,t1,fp1,n1);
                build_file_ptrs2 (ary,t2,fp2,0);
                for (n2=0; n2<t2->num_tp; n2++)  {
                        ONCE_FL = 0;
                        for (k=0; strcmp(ary[k],"^") != 0;
                             k++) {
                                fscanf (fp1[k],"%s",s1);
                                fscanf (fp2[k],"%s",s2);
                                if (strcmp(s1,s2) != 0)
                                        ONCE_FL = 1;
                                j = ftell (fp1[k]) -
                                   strlen(s1);
                                fseek (fp1[k],j,0);
                                j = ftell (fp2[k]) -
                                   strlen(s2) + t2->tpl_sz;
                                fseek (fp2[k],j,0);
                        }
```

```
                        if (!ONCE_FL)
                                add_tuple
                                (t1,t2,ary,n1,n2,fp);
                }
                for (k=0; strcmp (ary[k],"^") != 0; k++)  {
                        fclose (fp1[k]);
                        fclose (fp2[k]);
                }
        }

        t2->fp = fp;
        t2->tpl_sz = hdr_len;
        j = ftell (fp);
        t2->num_tp = (j/t2->tpl_sz) - 1;
        strcpy (t2->rel_name,str);
        fclose (fp);
/*
        check_prnt (4,1,1);
*/
}
```

```
/*********************************
*                               *
*        CREATE_JOIN_HDR        *
*                               *
*********************************

        This function copies the needed attribute names to
the header of the new relation that is to be formed and
updates the atr array.
Input:
ary -     array containing all key attributes of new rel
t1 -      ptr indicating position of rel 1 in list
t2 -      ptr indicating position of rel 2 in list
fp -      ptr pointing to new relation that is created
hdr_len - tuple size of the new relation
*/

create_join_hdr (ary,t1,t2,fp,hdr_len)
Atr_name     ary[];
Temp_nd_ptr  t1,t2;
FILE         *fp;
int          *hdr_len;

{
        int       i=0,j=0,k,J_ATR;
        Atr_name  str, aa[2*MAX_NUM_ATTRS_IN_REL];
```

```
        while (strcmp(t1->atr[i],"^") != 0)   {
                fprintf (fp,"%-10s",t1->atr[i]);
                strcpy (aa[j++],t1->atr[i]);
                i++;
        }
        i = 0;
        while (strcmp(t2->atr[i],"^") != 0)   {
                J_ATR = check_if_join_atr (t2->atr[i],ary);
                if (!J_ATR)  {
                        fprintf (fp,"%-10s",t2->atr[i]);
                        strcpy  (aa[j++],t2->atr[i]);
                }
                i++;
        }
        fprintf (fp,"\n");
        *hdr_len = ftell (fp);
        for (i=0; i<j; i++)
                strcpy (t2->atr[i],aa[i]);
        strcpy (t2->atr[i],"^");
}
```

```
/*********************************
*                               *
*        BUILD_FILE_PTRS2       *
*                               *
*********************************

        This function is used to build file pointers to the
attributes of the temporary relation pointed to.
Input:
ary -     array containing attributes in query
fpa -     array containing ptrs to attributes in new
          relation
n -       integer indicating the tuple number in the rel.
*/

build_file_ptrs2 (ary,p,fpa,n)
Atr_name     ary [];
Temp_nd_ptr  p;
FILE         *fpa[];
int          n;

{
        int       i,j;
        Atr_name  atr,rel_name;

        strcpy (rel_name,p->rel_name);
        for (i=0; strcmp(ary[i],"^") != 0; i++)   {
```

```
                fpa[i] = fopen (rel_name,"r");
                strcpy (atr,"^");
                while (strcmp(atr,ary[i]) != 0)
                        fscanf (fpa[i],"%s",atr);
                j = ftell (fpa[i]) - strlen(atr) +
                    ((n+1) * p->tpl_sz);
                fseek (fpa[i],j,0);
        }
}
```

```
/*********************************
*                                *
*           ADD_TUPLE            *
*                                *
**********************************

        This function is for adding a new row to the
relation that is created by the join of two existing
relations.
Input:
t1 -   ptr indicating position of 1st join rel in list
t2 -   ptr indicating position of 2nd join rel in list
ary -  array containing names of join attributes
n1 -   integer indicating tuple # in relation 1
n2 -   integer indicating tuple # in relation 2
fp -   file ptr pointing to new relation (got from join)
*/

add_tuple (t1,t2,ary,n1,n2,fp)
Temp_nd_ptr  t1,t2;
Atr_name     ary[];
int          n1,n2;
FILE         *fp;


{
        int       i,k,J_ATR;
        long      j1,j2;
        char      ch,line [MAX_TPL_LEN];
        Atr_name  s1,s2,atr;
        FILE      *f1,*f2;

        strcpy (s1,t1->rel_name);
        strcpy (s2,t2->rel_name);
        f1 = fopen (s1,"r");
        f2 = fopen (s2,"r");
        j1 = (n1+1) * t1->tpl_sz;
        fseek (f1,j1,0);
```

```
        fgets (line,MAX_TPL_LEN,f1);
        line [strlen(line)-1] = '\0';
        fprintf (fp,"%s",line);

        j1 = 0;
        while ((j1+10) < t2->tpl_sz)  {
                fseek  (f2,j1,0);
                fscanf (f2,"%s",atr);
                j1 = ftell (f2);
                J_ATR = check_if_join_atr (atr,ary);
                if (!J_ATR)  {
                        j2 = j1 - strlen (atr)
                             + ((n2+1) * t2->tpl_sz);
                        fseek   (f2,j2,0);
                        fscanf  (f2,"%s",atr);
                        fprintf (fp,"%-10s",atr);
                }
        }
        fclose  (f2);
        fprintf (fp,"\n");
}




/**********************************
*                                *
*        CHECK_IF_JOIN_ATR       *
*                                *
**********************************

        This function checks whether the given attribute is
among the attributes that are common to the join. If so it
returns a TRUE value.
Input:
atr -    attribute that is to be checked
ary -    array containing names of join attributes
*/

check_if_join_atr (atr,ary)
Atr_name   atr,ary[];

{
        int   i=0,DONE=0;

        while (!DONE)  {
                if (strcmp (atr,ary[i]) == 0)
                        DONE = 2;
                if (strcmp (ary[i],"^") == 0)
                        DONE = 1;
                i++;
```

```
        }
                if (DONE == 2)
                        return (1);
                else
                        return (0);
}
```

```
/*********************************
*                               *
*          PRINT_RESULTS        *
*                               *
*********************************

        This function is used to do the final projection of
the needed result attributes from the last temp rel created.
Input:
q_ary -   array containing the result attributes that are to
          be finally projected*/

print_results (q_ary)
Q_ary   q_ary;

{
        Temp_nd_ptr  t1;
        FILE         *fp[MAX_NUM_ATTRS_IN_REL];
        int          i,n1,n2;
        long         j;
        Atr_name     atr;

        printf ("\nResult:\n");
        printf ("=======\n\n");
        for (i=0; strcmp(q_ary[i],"^") != 0; i++)  {
                printf ("%-10s",q_ary[i]);
                fp[i] = fopen (t_hd->rel_name,"r");
                strcpy (atr,"^");
                while  (strcmp(atr,q_ary[i]) != 0)
                        fscanf (fp[i],"%s",atr);
                j = ftell(fp[i]) - strlen(atr) +
                    t_hd->tpl_sz;
                fseek (fp[i],j,0);
        }
        printf ("\n");
        for (n1=0; n1<i; n1++)
                printf ("----------");
        printf ("\n");

        for (n1=0; n1<t_hd->num_tp; n1++)  {
                for (n2=0; n2<i; n2++)  {
```

```
                    fscanf (fp[n2],"%s",atr);
                    printf ("%-10s",atr);
                    j = ftell(fp[n2]) - strlen(atr) +
                        t_hd->tpl_sz;
                    fseek (fp[n2],j,0);
                }
                printf ("\n");
        }
}
```

```
/*********************************
*                               *
*           CHECK_PRNT          *
*                               *
*********************************

        This is just a debugging routine. Performs no
function, unless used for checking intermediate results.*/
Input:
typ -     to indicate which structure to print
val1 -    maxm # of tuples (if relevant)
val2 -    maxm # of values (if relevant)
*/

check_prnt (typ,val1,val2)
int  typ,val1,val2;

{
        int             i,k=0;
        List_nd_ptr     t1;
        Temp_nd_ptr     p1;
        P_nd_ptr        t2;
        C_nd_ptr        t3,t4;

        switch (typ) {
        case 1 : {
        printf ("\nPrinting contents of matrix\n\n");
        for (i=0; i<val1; i++)   {
                printf ("[%d] %5s -",i,matrix[i].rel_name);
                k = 0;
                while (strcmp(matrix[i].attr_ary[k],"^") !=
                       0)
                        printf
                        ("%7s",matrix[i].attr_ary[k++]);
                k = 0;
                printf ("%7s",":");
```

```
        while (strcmp(matrix[i].key_ary[k],"^") !=
                0)
                 printf
                 ("%5s",matrix[i].key_ary[k++]);
        printf ("\n");
        printf ("Tp sz = [%d], and # of tpls =
                  [%d]\n", matrix[i].tpl_sz,
                  matrix[i].num_tp);
}
}
break;

case 2 :  {
printf ("\nq_ary    :    ");
for (i=0; i<val1; i++)
        printf ("%s  ",q_ary1[i]);
printf ("\n");
printf ("c_ary    :    ");
for (i=0; i<val2; i++)   {
        printf ("%s ",c_ary1[i].lhs);
        printf ("%s ",c_ary1[i].mid);
        printf ("%s;  ",c_ary1[i].rhs);
}
printf ("\n");
}
break;

case 3 : {
printf ("\nrel_list\n");
printf ("--------\n");
t1 = head;
while (t1 != NULL)  {
        printf ("rel_name   : %s\n",t1->rel_name);
        t2 = t1->p_ptr;
        printf ("p_ptr  ");
        while (t2 != NULL)   {
                printf (" --> %s",t2->data);
                t2 = t2->next;
        }
        t3 = t1->c_ptr;
        printf (" --|\nc_ptr  ");
        while (t3 != NULL)   {
                printf (" --> %s %s %s",
                        t3->data.lhs,t3->data.mid,
                        t3->data.rhs);
                t3 = t3->next;
        }
        printf (" --|\n\n");
        t1 = t1->next;
}
}
break;

case 4 :
```

```
        p1 = t_hd;
        printf ("Temp-rel's that are not empty are : \n");
        while (p1 != NULL)  {
                printf (" %-5s with atr.'s  :",p1-
                        >rel_name);
                i = 0;
                while (strcmp (p1->atr[i],"^") != 0)
                       printf (" %s",p1->atr[i++]);
                printf ("\n");
                printf ("tpl_sz  = %d\n",p1->tpl_sz);
                printf ("num_tp  = %d\n",p1->num_tp);
                p1 = p1->next;
        }
        printf ("\n  -*-  \n");
        break;

        }
}
```

---oooOOO END OF PROGRAM OOOooo---

VITA

FEROZE KHALIFULLAH

Candidate for the Degree of

Master of Science

Thesis: A QUERY OPTIMIZATION TECHNIQUE IN RELATIONAL DATABASES

Major Field: Computer Science

Biographical:

Personal Data: Born at Tiruchi, India, on March 31, 1965, to Mr. and Mrs. Kutbuddin Khalifullah.

Education: Graduated from St. Johns Vestry Higher Secondary School, Tiruchi, India, in April 1983; Received Bachelor of Engineering Degree with a major in Civil Engineering from P.S.G. College of Technology, Coimbatore, May 1987; Completed requirements for the Master of Science degree at Oklahoma State University in December 1991.

Professional Experience: Graduate Assistant, University Computer Center, OSU; Construction Engineer, Pavan Associates, Madras, India.