THE FRAGMENTATION PROBLEM

IN

DISTRIBUTED DATABASES

By

Nazih E. Khalil

A thesis

Submitted in Partial Fulfillment of
the requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science
Faculty of Natural and Applied Sciences

Notre Dame University – Louaize
Zouk Mosbeh, Lebanon

July 1999

THE FRAGMENTATION PROBLEM

IN

DISTRIBUTED DATABASES

Nazih E. Khalil

Approved:

Marie Khair, Assistant Professor of Computer Science.
Advisor.

Georges M. Eid, Professor of Mathematics,
Dean of Faculty of Natural and Applied Sciences.
Member of Committee.

Fouad Chedid: Associate Professor of Computer Science,
Chairperson of the Department of Computer Science.
Member of Committee.

Khaldoun El Khaldi, Assistant Professor of Computer Science.
Member of Committee.

Date of thesis defense: Tuesday July 6, 1999.

This thesis is dedicated to my family,
My parents Elias and Laurice,
My sister and her husband Lauretta and Alfred Maksoud,
My brother Sami.

ACKNOWLEDGMENT

For her patience and support, I am grateful to Dr. Marie Khair, Assistant Professor and my advisor in achieving this work.

I would like also to thank Notre Dame University, Faculty of Natural and Applied Sciences all its faculty and staff members, specially to Dr. Georges M. Eid, the Dean, Dr. Fouad Chedid, Chairperson of the Department of Computer Science, and Dr. Jean Fares, Chairperson of the Mathematics Department for their encouragement and support.

Colleagues really helped me, I hope they know how much I value that help. In particular are Mr. Salam Chouery, my sister Mrs Loretta Khalil Maksoud and her husband Alfred, Dr. Fady Said and his wife Rita who provided me some of the needed references from the United States and Canada; Mr. Raymond Al Chabab, Mr. Dory Eid and Mr. Jean Mina who gave me comments on my thesis.

Nazih E. Khalil

ABSTRACT

Data fragmentation is a heuristic problem in the design of a distributed database. Its purpose is to maximize the locality of reference, minimize data access at remote site, and to decrease the number of disk accesses in the system. In this thesis, we review the design a homogeneous distributed database. As the design is a heuristic problem, we also review a number of algorithms suggested as solutions to the three types of fragmentation: vertical, horizontal or mixed. Next, we propose that a previously implemented routine entitled "a transaction-based vertical partitioning algorithm" can be implemented when fragmenting the database horizontally. Finally, a simulation of the proposed technique is presented.

An accepted paper, in the 10th International Conference and Workshop on Database and Expert Systems Applications (DEXA '99) held in the University of Florence - Italy, entitled "Availability and Reliability Issues in Distributed Databases Using Optimal Horizontal Fragmentation" by Khair M., Khalil N. and Eid D.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AAM ..... Attribute affinity matrix
AAG ..... Attribute affinity graph
AUM ..... Attribute usage matrix
BVP ..... Binary vertical partitioning
CAAM ..... Cluster attribute affinity matrix
CDB ..... Centralized database
CPAM ..... Cluster predicate affinity matrix
CPS ..... Complex physical structure
DBA ..... Database administrator
DBMS ..... Database management system
DDB ..... Distributed database
DDBMS ..... Distributed database management system
DDM ..... Distributed directory management
GDBA ..... Global database administrator
LDBA ..... Local database administrator
NOV ..... Non-overlapping partitions
OVP ..... Overlapping partitioning
PE ..... Partition evaluator
PAM ..... Predicate affinity matrix
PUM ..... Predicate usage matrix
RDB ..... Relational database
SQL ..... Structured Query Language

LIST OF ALGORITHMS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I- DEFINING THE PROBLEM

Designing a distributed database (DDB) is an optimization task requiring solutions to several interrelated problems including data fragmentation, data allocation and data access optimization remotely and locally [8,9,22]. Divide and conquer techniques are adapted to bring solutions to this problem.

Researchers provided suggestions to one or more of the mentioned problems namely data fragmentation, data allocation and data access optimization remotely and locally. The aim behind these suggestions is to improve the overall system performance. Data fragmentation, if well handled, achieves this goal by increasing the locality of reference, decreasing the data access at remote site and decreasing the number of disk accesses. Data fragmentation is a heuristic optimization problem which splits a database into parts known as fragments. The problem is to find an optimal number of fragments in order to keep the specified aim. Researchers provided a set of perspectives and algorithms as solutions to the three types of fragments vertical, horizontal or mixed.

A vertical algorithm having a complexity of $O(2^n)$, having n number of transactions, has been proven to provide an optimal number of fragments having a better performance compared to the other suggested algorithms [12]. We propose that the application of this algorithm can be implemented in case of a horizontal partitioning.

This thesis is composed of eight chapters. Chapter II is an evolution notion to the implementation of DDB systems. Chapter III focuses on the design methodology of data fragmentation. Chapters IV, V and VI review and discuss the different provided algorithms to the vertical, horizontal and mixed partitioning, respectively. Chapter VII is a proposed horizontal algorithm which provides an optimal number of fragments followed by a simulation of the algorithm. The conclusion and future work are the essence of chapter VIII.

CHAPTER II- PRELIMINARILY NOTIONS

2.1 Notion of Databases

2.1.1 Definition

Databases and database technology are having a great impact on the use of computers. Databases are incorporated in different fields such as business, engineering, medicine, law, education and others. As the importance of database is increasing, it is better to know the meaning of the term. A database is a coherent collection of data with some inherent meaning, i.e., recorded facts. Thus, a random assortment of data cannot correctly be referred to as a database. A database is designed, built and populated for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested. A database has some source from which data are derived, some degree of interaction with events in the real world, and an audience that is actively interested in the contents of the database [15,16,17].

2.1.2 Historical Background

Before the year 1960, data were read in a sequential order. Afterward, the non-sequential access became feasible. Between the 1960 and 1980, the entity relationship model was developed. During this era, the concept of data communication networks and centralized databases (CDB) were implemented using some commercial database management systems (DBMS) such as INGRES and the structured query language (SQL). After the 1980, the client/server architecture for DDB models was implemented. Also in this period, powerful commercial DBMS like DB2, Oracle, Informix were adopted. These architectures altered the known meaning of database that contains not only data but also a complete definition of the database. This definition is stored in the catalog. It contains information such as the structure, the type and storage format of each data item. The information stored in the catalog is called meta-data. Databases provide data independence, a separation between data

and programs, i.e., database access applications and data (including meta-data and data) are written independently. Further, a database should support multiple views of data. It typically has many users, each of whom requires a different perspective of view of the database. A view may be a subset of the database and/or may contain virtual data that is derived from the database but it is not explicitly stored. Obviously, it must allow multiple users to access the database at the same time and share the data. This accessing process is controlled by concurrency control software to ensure that if several users are trying to update the same data then it is achieved in a controlled manner. These characteristics are easily achieved through the use of a DBMS, a combination of the database and sets of software programs that handle features of the data such as creation, insertion, update, deletion, query, concurrency control according to a user request and data recovery [15,16,17].

Data communication topology, known as network, is required to access the data stored at remote sites. A computer network is a collection of autonomous computers that are capable of exchanging information. As the network topology is beyond the scope of this study, we are sufficient to this definition. A more detailed explanation of computer network topologies and their implementations can be found in [6,15,30].

2.1.3 Centralized Databases

CDB systems are first kinds of a database system using a network topology. Their components, accessed by remote terminals, are located at a single site. These components include secondary storage, the DBMS, and the data. Figure 1 is a typical CDB having five sites where the components are located at site 1. As CDB is a brand of database, not only it inherits all the characteristics of a database mentioned in the previous section, but also it has

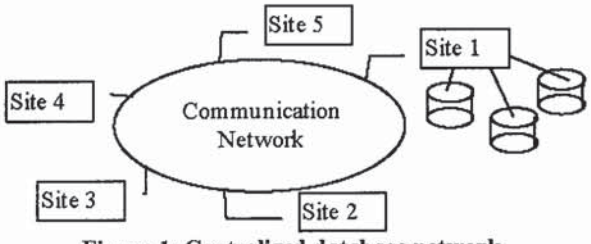

Figure 1: Centralized database network.

its own specifications. All users in a CDB environment access the latest update of the data. It controls redundancy by reducing the duplication of data as much as possible. It enforces consistency and integrity constraints on the database. Without forgetting the security constraints, CDB controls unauthorized accesses, i.e., the security on the database is maintained and supervised by a database administrator (DBA). Further, CDB provides backup and recovery of the database at least to the moment where the system has started [15,16,17,28].

2.2 Notions of Distributed Databases

DDB, a new trend of database, is evolving rapidly. It is a result of two opposite concepts: database systems and computer network technologies. As mentioned earlier the first states that data must be centralized. The second works against anything related to centralization. It should be clear that the connection between these two methodologies is the integration and not centralization. It is possible to achieve integration without centralization and this is what DDB system attempts to achieve [28].

2.2.1 Definition

There is no unique definition related to the concept of DDB. Authors [28] state that DDB is a collection of multiple, logically, interrelated databases that belong to the same system but are physically spread over (distributed) the sites of a computer network. Notice the given citation is vague. It emphasizes mainly the type of databases, their locations and the need of a computer network. The authors [28] have omitted other important components along with their interactions. Therefore, Ceri *et. al.* [6] elaborate the previous definitions and clarify that a distributed database is a collection of related data which are distributed over different sites of a computer

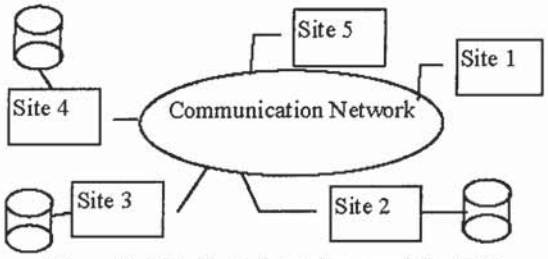

Figure 2: Distributed database architecture.

network. Each site of the network has autonomous processing that is capable of performing local applications. Furthermore, each site participates in at least one global application, which requires accessing data at several sites connected via a network topology [3,6,28].

Notice the word 'distributed' is repeatedly used because it is important to distinguish a DDB and a collection of files located at different locations. DDB is not a 'collection' of files that can be stored at each site of a computer network. To form a DDB, data should be logically related to the same system where the relation is defined according to a structural schema e.g, relational model (Figure 2) [26,28]. Furthermore, distribution does not mean that sites are geographically dispersed, processing units might be located in the same room each having its own database connected via a network. If the database is located in a single processing unit, the system becomes a CDB (Figure 1). Distribution is not restricted only to data but also it involves the processing logic, the function of hardware and software, and the control. In a word, distribution applies to the design and manipulation of the database [3,6,7]. On the other hand, several models were suggested for the file problem to minimize the overall cost function subject to the available secondary storage at each site. Further details on the file allocation problem can be found in [1,11].

Two types distinguish the DDB: homogeneous and heterogeneous. The first type resembles a CDB but instead of having the data stored at one site, they are distributed across a number of sites in a network. The homogeneous design is developed using the top-down model. The second type, heterogeneous DDB being as a recent trend is derived from autonomous preexisting databases stored under different types of DBMS. The latter design implements the bottom-up model. Such models have a certain degree of homogeneity indicating the degree to which individual DBMS can operate independently [3,15]. It is worth to mention that this study focuses on the first type, the homogeneous DDB.

2.2.2 Characteristics of Distributed Databases

Many aspects encourage the implementation of a DDB system due to the complex problems that the CDBs are facing [1,3,6,15,26,27,28]. Nowadays, organizations are naturally distributed over different locations. They keep on growing and becoming decentralized. Thus implementing a DDB approach fits more naturally the structure of the organization than a centralized system. The distribution aspects affect the system in its different components including processing logic, the function, the control and the data. The purpose is to improve the overall performance of the system.

Performance is achieved through a high degree of parallelism because of the existence of autonomous processors. Autonomy means that all operations at a given site are controlled by that site; no site X depends on some other site Y for its successful functioning. It also implies that local data is owned and managed by the local DBA. All the data belong to the local database even if it is accessible from remote site [15]. Bell *et. al.* [3] expand this concept and introduce four different types of autonomy. The design autonomy shows the design and implementation of local sites (data, model, and storage structure). The participation autonomy concerns how each site participates in a network topology and what is the data to share. Such decisions are issued from the local sites independently. Communication autonomy, dominated by the computer network, specifies the conditions to communicate with other sites. The execution autonomy controls the basic local operations (update, retrieve and insert) along with the abort operation of global transactions, which conflict with the local ones.

It is clear that when parallelism is adopted, it also reduces the communication overhead compared to centralized systems. Thus the communication cost is minimized because processing is achieved locally. Further on the economic level, installing in a system,

small computers with an equivalent power of a single big machine reduces more the overall cost of the DDB system [1,3,6,15,27,28].

The increase of availability and reliability of the system improves the overall performance. This is achieved by replicating the same fragment on different sites. Both properties have the goal to keep the system performing its task against different kinds of failures as opposite to CDB. In the latter, if the main site fails, all the system goes down but such cases rarely happen.

A good DDB design facilitates expandability with less impact on any existing applications. Expandability refers to the smooth incremental database growth of an organization. This is not easily feasible in a CDB environment unless the design from the beginning has foreseen this expandability otherwise a major impact will occur not only on the new applications but also on the existing ones. In brief, implementing DDB solves big and complex problems faced today by the CDB using the divide and conquer rule.

The degree of distribution transparency is another characteristic in a DDB system. A distribution transparency refers to the separation of the higher-level semantics of a system from the lower-level implementation issues. In other words, a transparency system hides the implementation details from users because programs are written as if the data is not distributed. There exist different types of transparencies [6,15,26,27,28]. Fragmentation transparency is when the user or application programmer works on the global relations as if the data is not fragmented. The local mapping transparency refers to several problems in distributed DBMS (DDBMS) without having to take into account the specific data models of local DBMS. This problem of transparency exists mainly in heterogeneous DDBMS where it is reduced in the homogeneous systems. The location transparency requires the user or application programmer to work on fragments regardless of their locations instead of global relation. Of course in DDB, some fragments are replicated to increase the reliability and

availability of the system. Replication transparency hides this duplication of fragments from the user. Surely, site failures in a DDB system occur in a way or another. The system, in such cases, is able to detect a failure, to reconfigure itself so that the processing may continue and execute the necessary recovery when the link is repaired. The concurrency transparency shows the concurrent transactions as if they are processed in a serial order. Because users are manipulating (updating) the data on different sites, data integrity has a high potential degree to be violated especially when the system runs many transactions concurrently. No matter which type is referred to, transparency hides the distribution features of the DDB from the user. The system is revealed to him as if it is a CDB.

Even though DDB is very useful, it has also many disadvantages. Researchers lack the experience for such development. DDB design has new problems added to the unsolved ones of CDB. Beginning with the fragmentation of data, this issue is discussed in the coming sections of this paper due to its importance. Next comes the distributed concurrency control. Its role is maintaining not only the database integrity by synchronizing the accesses to the DDB but also the coordination among sites. Since data is dispersed on several sites, security stays more problematic. Local security is implemented with the same degree of efficiency similar to the centralized system. Global security merely exists due to the autonomy of each site. If the global security is high and efficient then there will be no local independence and the system will be considered as a CDB. From a different point of view, the distributed query processing will be slow in performance and the cost of communication increases as long as the data is not found on the lookup site. Another disadvantage of DDB is the distributed directory management (DDM) which keeps tracks of the data. The DDM by itself may be global or local at each site, one or multiple copies, and distributed over several sites or stored on a single location. Further, the distributed deadlock management is the same as for the centralized operating system resulting from the competition among users to access the same

data. For the heterogeneous DDB system, a translation mechanism is also needed. This usually occurs when designing a DDB system from existing centralized ones. It should be clear that the above-provoked problems are considered as interconnected instead of independent issues (Figure 3) [6,26,27,28].

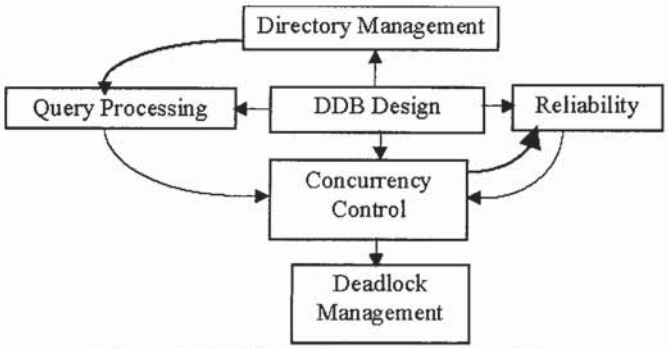

Figure 3: Relationship among DDB modules

2.2.3 Distributed Database Management System Concept

It is necessary to have an also idea about DDBMS which handles a DDB. A DDBMS controls, supports and manages the creation and maintenance of DDB. In addition to this, it coordinates the access to data at various sites knowing that each site may have its own DBMS to manage its local data. Many types of DDBMS are determined. These types are ordered in a tree form shown in Figure 4.

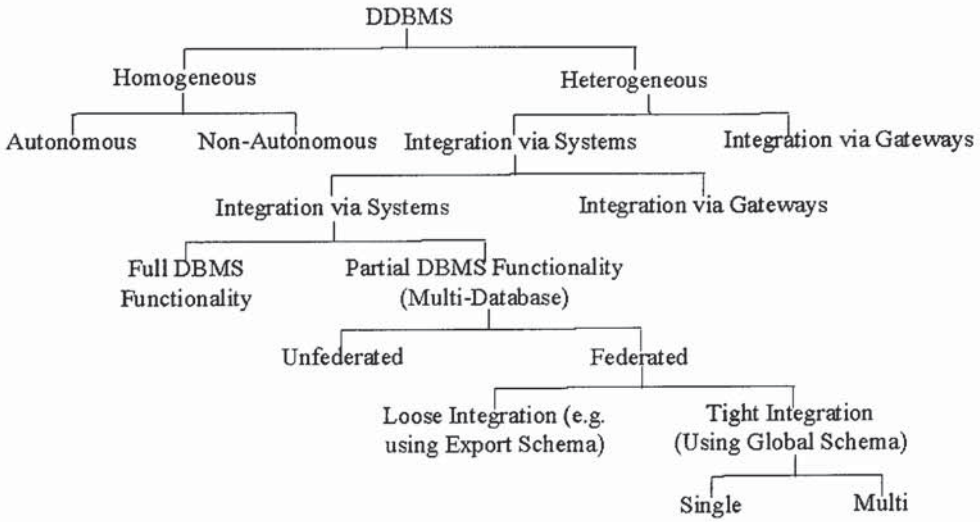

Figure 4: Taxonomy of distributed data systems.

DDBMS provides additional features to the DBMS of a CDB. The ability to send transactions and queries to remote sites and access the data in more than one site in order to accomplish the request. Next, the ability to keep track of the data distribution and replication in the DDBMS catalog and to decide which replication copy to use and to maintain the consistencies of the copies and which copy to access. Also, the DDBMS must have the ability to recover from site or other failure and make the necessary updates. On the hardware level, multiple computers (considered as sites) and using a communication network distinguishes it from a centralized one. Figure 5 on the next page is a typical DDBMS architecture. Several commercial database management are developed and even used in the market to accomplish the needs of a DDB systems such as SDD-1, R*, BUBBA, DB2, INGRES and ORACLE. In brief, A DDBMS is characterized by local autonomy, location transparency, fragmentation and replication transparency, no central site, i.e., no DBMS on a site is more important than another and continuous operation meaning that no planned activity should require a shutdown. Also a DDBMS must provide data management functions such as security, concurrency control, deadlock control, query optimization and failure recovery. Also there exist independence among hardware, operating system, network and the DBMS. Without forgetting, it should translate the request from one site using a local DBMS to another site having a different local DBMS [3,6,15,20,26].

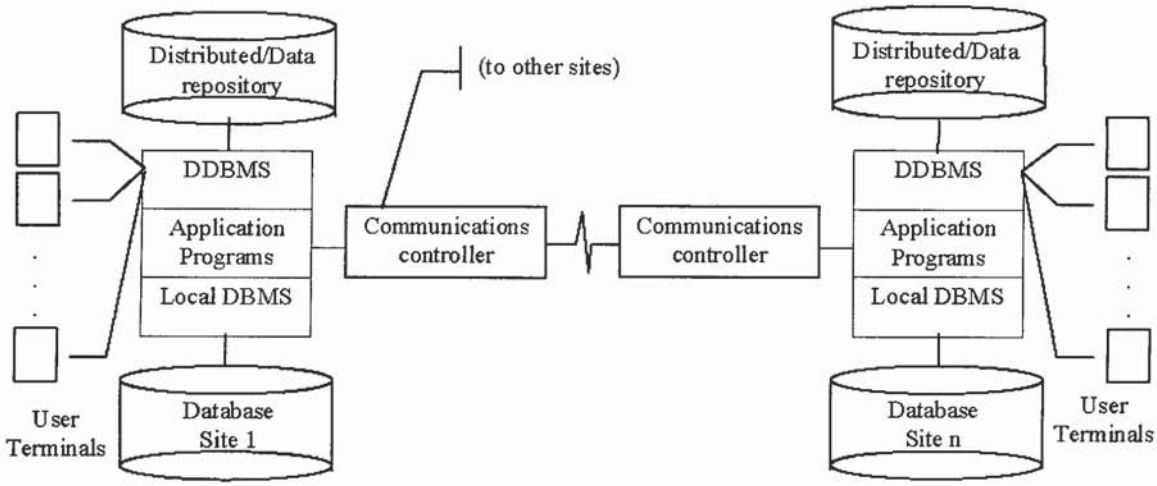

Figure 5: A typical DDBMS Architecture.

2.3 Comparison between Centralized and Distributed Databases

As a conclusion of this section, Table 1 on the next page emphasizes on some of the important differences between CDB and DDB that I have grouped.

Feature/Reason	CDB	DDB
Organization reasons	Becomes questionable	Fits naturally
Expandability (database incremental growth)	Difficult to achieve without having an impact on existing applications and sometimes unfeasible	Easy to implement with almost no impact on existing applications
Interconnection between existing databases	Difficult to achieve. It is preferable to consider creating a new centralized system taking into consideration the existing databases	Just use bottom-up design
Location of data	Located at one Site	Dispersed over several sites
Communication overhead	High, very dependant on the network	Low, reduced due high degree of parallelism
Centralized control	One control to all information of a firm by a DBA	Opposes the centralization concept. GDBA is almost missing. LDBA create high degree of autonomy.
Data independence	The actual organization of data is transparent to the application programmer: "Conceptual schema."	Applied but add to it distribution transparency.
Redundancy	Reduced as much as possible: having one copy to reduce the inconsistencies of the same logical data, saving storage space.	Desirable feature: it makes the system available through replication. Availability of the system which does not stop processing if a site fail. DDB is available if data is replicated.
Complex physical structure and efficient access	Secondary indexes, interfile chains are a major part of the DBMS. Through it, obtain efficient access to the data	Need different tools: distributed access plans include also global and local optimization. It is not the right tool to achieve efficient access to the data.
Integrity, recovery and concurrency control	Transactions are the solution of these problems	Act as if the system is centralized.
Privacy and security	Depending on the DBA that controls the data accesses. Control procedures are needed.	LDBA + high degree of autonomy enforce more their own data protection. Communication problem is a weak point.
Failures	Rarely	Frequent site failures.
Reliability & Availability	Mainly through the use of Backup and Security issues	It is achieved by replicating the data.
Reality implementation	Exist since the 80s	During the 80s, it existed in prototypes. Now, commercial DDBMS are on the market like ORACLE, Distributed Ingres, and others…

Table 1: Comparison between DDB and CDB.

CHAPTER III- FRAGMENTATION DESIGN METHODOLOGY

DDB design can be achieved in two ways. The homogeneous (top-down) design processes beginning with the requirement analysis and logical design to the conceptual design and reaching the physical design on each site. In the case of heterogeneous DDB (Figure 4), the process is achieved bottom-up design, which involves the integration of existing database to reach a conceptual schema [6,27].

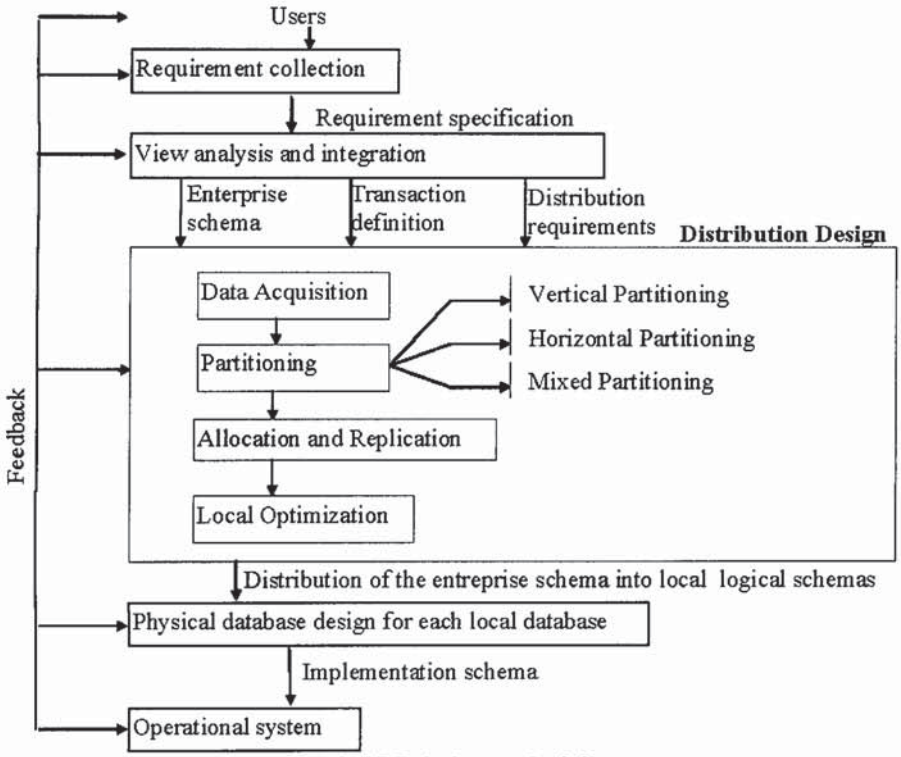

Figure 6: DDB design methodology.

This thesis focuses on the homogeneous design. Figure 6 sketches an overall top-down distribution methodology [4,8,9,22]. The user requirement is collected and analyzed. The views are integrated from the requirement specification [15]. Three inputs are required to achieve the distribution design phase. The enterprise schema known also as the global schema [6,28] describes the relation as if the database is not fragmented. The transaction definition is concerned with the frequency of transactions to be processed in the DDB design. And the distribution requirement emphasizes how the data is needed (distributed) knowing

that the user must have a good idea how to achieve it. The physical DDB design for each local database is how to organize the database locally. All these mentioned phases are processed in the same manner as in CDB. The distribution design of Figure 6 is an optimization problem that requires understanding and solutions to several interrelated problems including data fragmentation, allocation and local optimization. Data fragmentation clearly identified in Figure 6 is a problem in the design of DDB [4,7,8,9,22].

Other methodologies are suggested including synthesized extended entity relationship with a distributed transaction scheme known as SEER-DTS [10] and the semantic meta-model [2]. The SEER is an extension of the entity relationship model. Its modeling is object-based. The DTS is transaction modeling based on earlier modeling. Thus the SEER-DTS combined together, results and facilitates the DDB design schema design. The semantic meta model for DDB is suggested. Its purpose is to achieve fragmentation using the SQL. The authors assume a command statement:

FRAGMENT <name of fragment> IS <relation>
[WHERE <condition>] |
[WHERE <var> IN SELECT statement]

Different alternatives have suggested as a way to solve the distribution design phase. In [1], Apers considered data fragmentation and allocation as a single problem and it must be solve accordingly. In [9], Charkavarthy *et. al.* considered the alternative where data fragmentation is directly followed by allocation and replication, thus these issues are loosely coupled. These two alternatives are not the only ones; others were taken into account by many resarchers but are less important in this thesis. Considering the loosely coupled case, the definition of data fragmentation can now be easily established. Data fragmentation is the process of dividing global relations into subsets known as fragments which are classified into three categories: the horizontal fragments, the vertical fragments and the mixed fragments.

Then these fragments are distributed over different sites [7,23]. Data fragmentation also increases the locality of reference so that the overall system performance is improved [21] and decreases the number of disk accesses in the system [12,13,14]. Furthermore, while (or after) fragmenting a database two important features must be maintained. The completeness ensures that no data item is lost. Data items means a vertical domain, a horizontal tuple or a simple item value of the database as a result of a mixture of the last two ones. This is somehow a difficult task and needs more attention when considering the mixed partitioning. The reconstruction feature insures that the global relations must be reconstructed based on the given the fragments [2,16,21]. But the question remains how to fragment while/after maintaining these two features? How to create an optimal number of fragments in order to improve the system performance through the increase of locality of reference and the decrease of remote data access? Researchers have sets of algorithms that deals with each of the mentioned of types of fragmentation.

CHAPTER IV- THE VERTICAL FRAGMENTATION TECHNIQUES

This chapter begins with the identification of the vertical partitioning. Next, an overview of suggested algorithms for solving this issue is presented. The chapter is concluded with a summary that I have grouped showing the relationships among different algorithms and another summary showing the most important criteria in the studied algorithms.

4.1 DEFINITION

Vertical fragmentation is the projection of the global relations onto subsets of attributes, which form the fragment. However, once these relations are vertically partitioned, their reconstructions are impossible to achieve and thus a complete tuple cannot be obtained unless the primary key of each relation (or a system identifier) is appended to each fragment.

Many algorithms are suggested for dividing attributes into fragments [15,16,20,28].

4.2 OVERVIEW

In [23], a binary vertical partitioning is implemented based on an empirical objective function. It begins by selecting the transactions having a high degree of occurrences and their respective attributes that are grouped in a matrix form called Attribute Usage Matrix (AUM) where the rows represent the transactions and the columns the attributes. The elements of the matrix are either 1 or 0 (Table 2). The value is 1 if transaction T_i is accessing attribute A_j otherwise it is 0. Two additional columns are added to the AUM. The first column is the type of operations that have the value *u* for Update or *r* for Retrieval. The second is the access column that shows the number of occurrences each transaction accessing the attributes (Table 2a) from a single site. In case of multiple-site, a site matrix is used instead of the access column of the mentioned table. This Table 2b has as rows the

respective transactions and as columns the sites from where these transactions are issued. The elements of this matrix are how many times a transaction T_i is issued from site S_k.

(a) Attribute Usage Matrix

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	Ty	Access
T1	1	0	0	0	1	0	1	0	0	0	r	25
T2	0	1	1	0	0	0	0	1	1	0	r	50
T3	0	0	0	1	0	1	0	0	0	1	r	25
T4	0	1	0	0	0	0	1	1	0	0	r	35
T5	1	1	1	0	1	0	1	1	1	0	u	25
T6	1	0	0	0	1	0	0	0	0	0	u	25
T7	0	0	1	0	0	0	0	0	1	0	u	25
T8	0	0	1	1	0	1	0	0	1	1	u	15
Attribute length												
	10	8	4	6	15	14	3	5	9	12		

(b) Site Matrix

	S1	S2	S3	S4
T1	10	15	0	0
T2	10	20	10	10
T3	0	0	15	10
T4	10	15	0	10
T5	5	10	5	5
T6	10	5	5	5
T7	5	10	5	5
T8	5	5	2	3

Table 2: A typical attribute usage matrix and a site matrix.

The difference between using access column and site matrix is in the calculation of the attribute affinity matrix (AAM) which a symmetric matrix having for rows and columns the attributes (Table 3). The calculated number is known as the affinity value that shows how strong is the bond among a pair attributes. The affinity number is calculated using the equation

$$aff_{ij} = \sum_{k|u_{ki}=1 \wedge u_{kj}=1} acc_k \qquad (1)$$

where aff_{ij} is the affinity value among two attribute a_i and a_j. It is the summation of accesses of all transactions accessing the respective two attributes. u_{ki} and u_{kj} are the elements in the AUM for transaction k and attribute i and j respectively; u_{ki}=1 if transaction k uses attribute a_i, else it is equal to 0, similarly to u_{kj}. acc_k is the number of accesses of transaction k accessing both for attributes i and j. For a single site, $acc_k = n_k freq_k$ (n_k is the number of accesses to object instances for one occurrence of transaction k, $freq_k$ is the frequency of occurrence of transaction k), for example in Table 2a, $acc_1 = 25$ for k=1, $n_k = 1$ and $freq_k = 25$ and so on. For multiple sites, $acc_k = \sum_j n_{kj} freq_{kj}$ (n_{kj} is the number of accesses to object instances for one occurrence of transaction k at site j, $freq_{kj}$ is the frequency of

occurrence of transaction k at site j). For example in Table 2, acc_1=10+15+0+0=25 where $n_{kj}=1$ for all k and j and therefore the matrix site is converted to a single access column in the purpose of calculating the AAM. For example, the AAM of the AUM (Table 2a) using the access column is shown in Table 3.

	1	**2**	**3**	**4**	**5**	**6**	7	**8**	**9**	**10**
1	75	25	25	0	75	0	50	25	25	0
2	25	110	75	0	25	0	60	110	75	0
3	25	75	115	15	25	14	25	75	115	15
4	0	0	15	40	0	40	0	0	15	40
5	75	25	25	0	75	0	50	25	25	0
6	0	0	15	40	0	40	0	0	15	40
7	50	60	25	0	50	0	85	60	25	0
8	50	60	25	0	50	0	85	60	110	75
9	25	75	115	15	25	15	25	75	115	15
10	0	0	15	40	0	40	0	0	15	40

Table 3: An attribute affinity matrix

	5	**1**	7	**2**	**8**	**3**	**9**	**10**	**4**	**6**
5	75	75	50	25	25	25	25	0	0	0
1	75	75	50	25	25	25	25	0	0	0
7	25	50	85	60	60	25	25	0	0	0
2	25	25	60	110	110	75	75	0	0	0
8	25	25	60	110	110	75	75	0	0	0
3	25	25	25	75	75	115	115	15	15	15
9	0	25	25	75	75	115	115	15	15	15
10	0	0	0	0	0	15	15	40	40	40
4	0	0	0	0	0	15	15	40	40	40
6	0	0	0	0	0	15	15	40	40	40

Table 4: A clustered attribute affinity matrix

Next the AAM is clustered grouping the high affinity values together in an upper left block and smaller affinity values in the lower block. Thus the clustered attribute affinity matrix (CAAM) is a semi block diagram grouping the large values with large ones and small values with small ones. The grouping is achieved by implementing a CLUSTER algorithm, which permutes the rows and columns maximizing equation (2). The CAAM of AAM in Table 3 is Table 4.

$$\sum_{ij} aff_{ij}\left(aff_{i,j-1}+aff_{i,j+1}+aff_{i-1,j}+aff_{i+1,j}\right) \qquad (2)$$

Now the partitioning process begins. There are two types of partitioning nonoverlapping and overlapping fragments. The nonoverlapping fragments are chosen by placing a cut point x on the CAAM along the main diagonal such that the objective function

$$z = CL \times CU - CI^2 \text{ is maximized} \qquad \textbf{(3)}$$

where $CL = \sum_{k \in LT} acc_k$, $CU = \sum_{k \in UT} acc_k$, and $CI = \sum_{k \in IT} acc_k$. Using $A(k) = \{i \mid u_{ki} = 1\}$ as a set of attributes used by transaction k. T={k|k is a transaction}, U represents the attributes in the upper block and L for lower block that are identified after the cut x on the CAAM is specified. There are $n-1$ possible positions for the cut point x_1 where n is the number of attributes (Table 6). The same process is achieved for the overlapping using two cut points x_1 and x_2 on the CAAM with a nonempty intersection and maximizing the same object function z (Table 6). In this case there are $\frac{n(n-1)}{2}$ possible positions on the matrix for the two cut points. The difference between a nonoverlapping and a overlapping fragmentation, shown in Table 5, is the evaluation sets parameters that are used to calculate CL, CU and CI, the parameters of the objective function.

Evaluation sets	**NON-OVERLAP**	**SPLIT_OVERLAP**
LT	$\{k \mid A(k) \subseteq L\}$	$\{k \mid (type_k = 'r' \wedge A(k) \subseteq L) \vee (type_k = 'u' \wedge A(k) \subseteq (L - I))\}$
UT	$\{k \mid A(k) \subseteq U\}$	$\{k \mid (type_k = 'r' \wedge A(k) \subseteq U) \vee (type_k = 'u' \wedge A(k) \subseteq (U - I))\}$
IT	$T - \{LT \cup UT\}$	$T - \{LT \cup UT\}$

Table 5: Evaluation sets for the overlapping and non-overlapping fragments.

	5	**1**	7	**2**	**8**		**3**	**9**		**10**	**4**	**6**
5	75	75	50	25	25		25	25		0	0	0
1	75	75	50	25	25		25	25		0	0	0
7	25	50	85	60	60		25	25		0	0	0
2	25	25	60	110	110		75	75		0	0	0
8	25	25	60	110	110		75	75		0	0	0
----	----	----	----	----	----	x_2	----	----	----	----	----	----
3	25	25	25	75	75		115	115		15	15	15
9	0	25	25	75	75		115	115		15	15	15
----	----	----	----	----	----	----	----	----	x_1	----	----	----
10	0	0	0	0	0		15	15		40	40	40
4	0	0	0	0	0		15	15		40	40	40
6	0	0	0	0	0		15	15		40	40	40

Table 6: BVP cut points for nonoverlapping and overlapping

A further research shows that these fragments can be allocated to physical sites by refining the objective function and adding cost evaluation constraints. Four types of costs constraints exist. C_1 is the cost of irrelevant attributes accessed within a fragment (per byte). C_2 is the cost of accessing fragments for retrieval and updates (per access). C_3 is the storage cost (per byte). And C_4 is the transmission cost (per byte). The objective function becomes $\min z = \sum_{1 \le i \le 4} w_i c_i$. C_i are the previously mentioned cost factors; w_i refers to the weight calculated according to:

	w_1	w_2
Multiple sites Nonreplicated Allocation	$\sum_k acc_k \sum_{F \in S_k} \sum_{(i \in F) \wedge (i \notin A(k))} l_i$	$CL + CU + 2 \times CI$
Multiple sites Replicated Allocation	Irrelevant	$\sum_{k \mid (type_k = 'u') \wedge (A(k) \cap F \neq \phi)} \sum_j n_{kj} freq_{kj}$

	w_3	w_4
Multiple sites Nonreplicated Allocation	$\sum_{i \in I} l_i$	$\sum_k \sum_{F \in S_k} \sum_{j \mid alloc_F \neq j} n_{kj} freq_{kj} \sum_{i \in (F \cap A(k))} li$
Multiple sites Replicated Allocation	$\sum_{i \in P} \lvert alloc_F \rvert \sum_{i \in P} l_i$	$\sum_{k \mid A(k) \cap F \neq \phi \, j \mid j \notin alloc_F} \sum n_{kj} freq_{kj} \sum_{i \in (F \cap A(k))} l_i$

Table 7: Calculating the weight for the objective function

Note that in Table 7, l_i is length of attribute i, $TL = \sum_i l_i$ is the total length of the object, $UL = \sum_{i \in U} l_i$ is the length of the attributes in the upper block. The set $S_k = \{L\}$ if $k \in LT$, or $S_k = \{U\}$ if $k \in UT$, or $S_k = \{L, U\}$ if $k \in IT$.

The authors of [13,14] extend the algorithm of [23] taking into consideration the physical aspects of minimizing the disk accesses and the type of scans each transaction is applying on the relation. The scans are obtained from the query optimizer DBMS. The calculation of the cost is based on the average number of disk accesses and tuple retrieved.

The expected number of tuple retrieved can be estimated as the cardinality of the relation. The number of disk accesses is related to the type of scan as shown in Table 8.

	Index scan		Segment scan
	Clustered	Unclustered	
# of disk access	$\frac{Cardinality \times Selectivity \times len\ of\ tuple}{page\ size}$	$Cardinality \times Selectivity$	$\frac{Cardinality \times len\ of\ tuple}{(page\ size) \times n}$

Table 8: Calculation of the cost index and segment scan

where n is the prefetch factor, i.e., number of pages read in each disk access. The selectivity is the average number of pages read by a particular type of transaction. An algorithm for vertical partitioning to minimize disk accesses is presented through the usage of decision variables X_{ij}=1 if attribute i is assigned on site j or 0 otherwise. The number of disk accesses $f(X_{ij})=T_1+T_2$ where T_1 is the number of disk accesses required to retrieve the scanned segment and T_2 is the number of disk accesses required to retrieve the remainder of the original tuples for those transactions which require it. N is the number of attributes in a relation. a_{j1} is the attribute has the smallest expected value of selectivity of type j transaction. If attribute k $\in$ the same segment $S/a_{j1} \in S \Rightarrow X_{a_{j1}1}X_{k1} + X_{a_{j2}2}X_{k2} = 1$. Equation (4) is the tuple size of the primary segment for type j transaction.

$$\sum_{k}^{N}\left(X_{a_{j1}1}X_{k1} + X_{a_{j2}2}X_{k2}\right)(size\ of\ attribute) + L \qquad (4)$$

where L is the length of tuple identifier or primary key. Equation (5) is the number of disk accesses to retrieve the segment scanned per scan.

$$T_1^j = \frac{(tuple\ size)(cardinality)}{(page\ size)(prefetched\ blocking\ factor)} \qquad (5)$$

And equation (6) is for the index scan.

$$T_1^j = cardinality \times selectiviy\ of\ scanned\ attribute \qquad (6)$$

The equation (7) is the number of disk accesses to retrieve attributes not in the primary segment being scanned.

$$T_2^j = A \times Cardinality \times Selectivity\ of\ scanned\ attribute \times \left(1 - \left(X_{a_{j1}1} X_{a_{j2}1} \ldots X_{a_{jN}1} + X_{a_{j1}2} X_{a_{j2}2} \ldots X_{a_{jN}2}\right)\right) \quad (7)$$

where A is the number of additional disk accesses. All the equations (4), (5) and (6) depending on the type of scan with equation (7) are used in equation (8), the objective function to be minimized.

$$\sum_j \lambda(j)\left(T_1^j + T_2^j\right) \quad (8)$$

where $\lambda(j)$ is the scanned frequency of type j transaction.

The divide and conquer algorithm of [7] is used to solve large and small problems. The divide implements the same techniques of [23]. Its purpose is to solve large problems for data fragmentation, data allocation using a simple cost evaluation scheme. The conquer tool (Figure 7) not only uses the same modules for large problems but also it takes into consideration the application optimization, the operation allocation and it has a detailed cost evaluation for smaller problems. In other words, The vertical fragmentation, data allocation and simple cost evaluation modules in Figure 7 are reviewed in [23]. The application optimization module selects the needed logical fragments to achieve the query and selects also the best operation tree among various equivalent ones. The operation allocation is concerned with the physical aspects of the fragment. It selects the physical fragments and the sites where each operation is executed. The results are evaluated upon the problem size. If the latter is large a simple performance is applied otherwise the evaluator module is processed (Figure 7), which increases the complexity of the vertical partitioning problem. At each process two subproblems are created and thus for each problem the same process is applied recursively until optimal solution is reached.

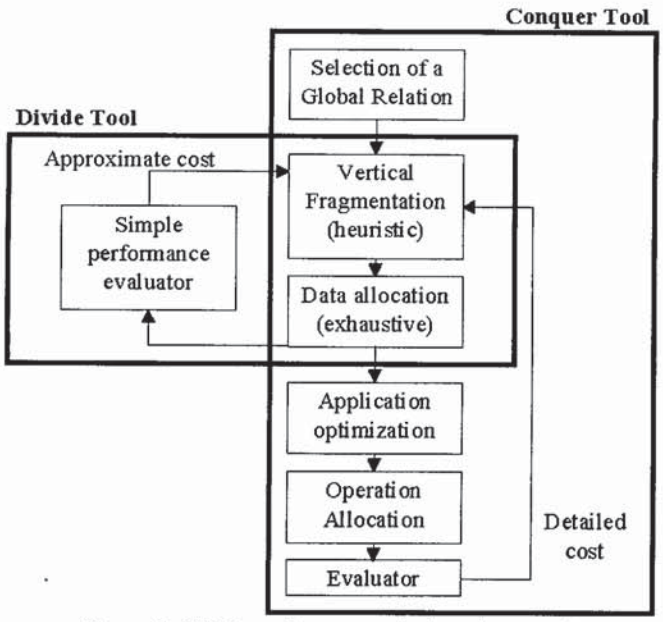

Figure 7: Divide and conquer tool environment.

In [25], a graphical vertical algorithm is developed. This algorithm has a less computation than the previous techniques. The algorithm uses an attribute affinity graph (AAG). It is a graph interpretation of the AAM. The nodes of the graph represent the attributes, whereas the edges are the non-zero affinity values. The zero values in the AAM are disregarded in the AAG. Thus, the graph of the affinity matrix of Table 3 is represented in Figure 8. Next, a linearly connected spanning tree is constructed. The purpose of the algorithm is to generate all meaningful fragments in one iteration by considering every cycle as a fragment. Before going into the steps of the algorithm, the authors define the following notations: A primitive cycle denotes any cycle in the affinity graph. Affinity cycle denotes a primitive cycle that contains a cycle node (unless otherwise mentioned every cycle is considered as affinity cycle). Cycle completing edge denotes a "to be selected" edge that would complete a cycle. Cycle node is that node of the cycle completing edge, which was selected earlier. Former edge denotes an edge selected between the last cut and the cycle

node. Cycle edge denotes any of the edges forming a cycle. Extension of a cycle refers to a cycle being extended by pivoting at the cycle node.

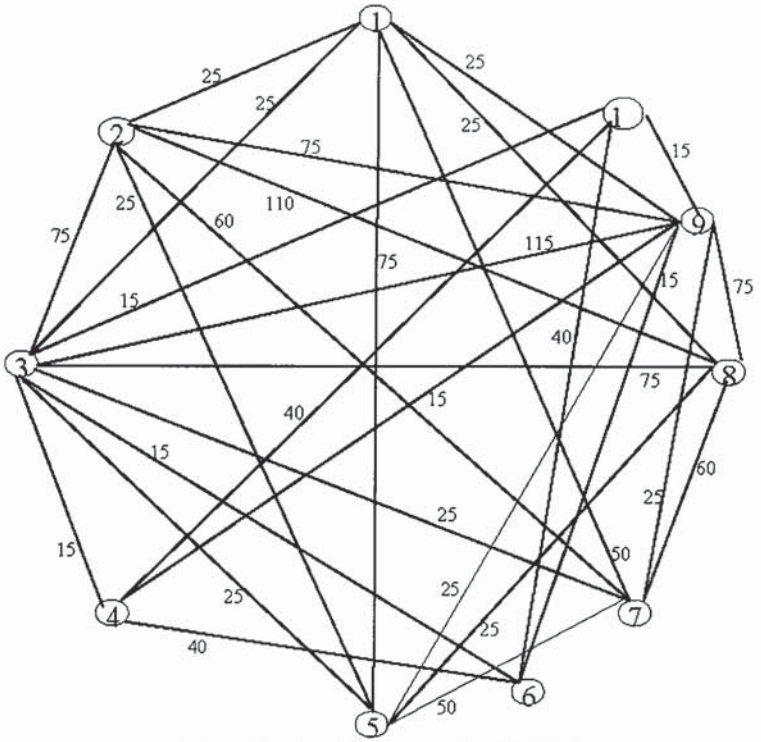

Figure 8: An affinity graph of Table 3

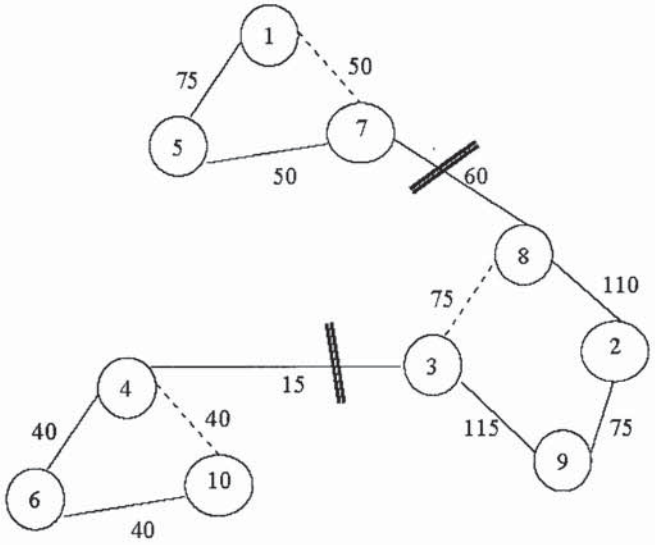

Figure 9: Result of the vertical graph algorithm of Figure 8

Algo 1 describes the graphical algorithm. The routine is achieved in 5 steps.

Construct the affinity graph from the affinity matrix. (Disregard the 0 edge value, and the main diagonal).

Start from any node.

Select an edge linearly connected to the tree and have the large value among the possible choices at each end of the tree.

The iteration will end when all nodes are used for tree constructions.

When the next edge selected form a primitive cycle.

- If a cycle node exists then disregards this edge, go to 3.
- If a cycle does not exist then look for the possibility of extension of a cycle? If okay mark the cycle as affinity cycle and consider it as a candidate partition, go to 3.

When the next edge selected does not form a cycle and a candidate partition exists.

If no former edge is found, check the possibility of extension? If also not found then cut this edge and let the cycle be a partition, go to 3.

If a former edge is found, change the cycle node, check the possibility of extension? If also not found then cut this edge and let the cycle be a partition, go to 3.

Algo 1: Vertical graphical partitioning

Creating all meaningful fragments in one-iteration is the efficiency of Algo 1. However, it has also deficiencies relative to its claim that states "all pairs of attributes in a fragments, have high affinity within fragment affinity but low between fragment affinity"[25]. According to [18,19] the deficiencies are that the produced output does not necessarily satisfy the claim: the Algo 1 generates solutions independent from the input order. A simple counter example supplied by [18,19] proved that what it has been said is true.

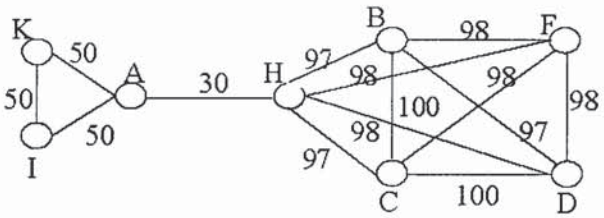

Figure 10: A counter example to Algo 1

When applying Algo 1 on the graph of Figure 10, the resulting fragments are $\{\{B,F,D,C,\},\{H\},\{A,I,K\}\}$. It is difficult to justify the fragment with only one attribute $\{H\}$ where it is easy to observe that the graph of Figure 10 produces two fragments $\{\{H,B,F,D,C\},\{A,I,K\}\}$. Therefore, a new graphical algorithm based on a clustering

technique is suggested in [18,19]. It replaces the unnecessary restriction of an affinity cycle [25] by the requirement of 2-connectivity.

Here are some basic definitions needed for the new graphical algorithm defined in [18,19]. An affinity graph is formed by a positive weighted graph noted as $G=(V,E,p)$ where V is the set of vertices and E is the set of edges. For each edge e, $p(e)$ is its affinity value. Also, in a graph (V,E), $(u,v)\in E$ is incident to u and v. A graph (V',E') is a subgraph of (V,E) if $V'\subseteq V$ and $E'\subseteq E$. Further, (V,E) is connected if for each pair u,v of distinct vertices, there is a path connecting u and v. (V,E) is 2-connected if $|V|>3$, and if after the removal of an arbitrary vertex, the resultant graph is still connected. Let $V'\subseteq V$. A subgraph (V',E') is induced by V' if E' is the set of those edges which are incident to a pair of vertices in V'. A connected component of (V,E) is a connected subgraph (V',E') such that for each vertex u of V' and each vertex v of $V-V'$ there is no path from u to v. A vertex u of (V,E) is isolated if there is no edge incident to u. Using these notations and settings, the authors of [18,19] defines a cluster V' of an affinity graph $G=(V,E,p)$ as a subset of V if either

(a) $|V'|>1$ and there is a connected subgraph (V',E') such that for each e of E', $p(e)$ is larger than the affinity values of those edges going out from V',(e.g. in Figure 11, (B,H), (C,B), (D,B) go out from $\{C,H,D\}$), or

(b) V' consists of an isolated vertex of G.

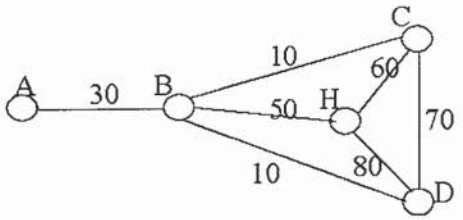

Figure 11: A connected graph example

The non-overlapping fragmentation (NOV) algorithm clusters (groups) the vertices of the affinity graph G (Algo 2). Each cluster is a subgraph S of G. The grouping technique is achieved in a manner that the affinities in S are greater than the ones existed on edges that are going out from S. For example in Figure 11, $\{A, B, C, D, H\}, \{B, C, D, H\}, \{C, D, H\}, \{D, H\}$

```
Algorithm NOV(G,var F)      /* G is a graph having V set of vertices and E set of edges */
Input G: affinity graph;
Output F: set of fragments;
Begin
   F= φ ; j=0; SORT(G,M,k) (M={Mi: 1 ≤ i ≤ k });
   V0=V;E0=M;
 While Vj ≠ φ do begin
   CLUSTER((Vi,Vi,p),T);
   OBTAIN(T,S(T));              /* return the affinity indices */
   CHOOSE(T,S(T),f);
   F = F ∪ f ; j=j+1;T= φ ; Vj = Vj-1-f;
   Delete the edges incident to a vertex of f from Ei-1 to form Ei;
 End
End
```

are clusters.

Algo 2: Non-overlapping partitioning

The NOV implements a slight modification of the CLUSTER algorithm by removing the SORT in step 1 of Algo 3. The procedure OBTAIN returns the $S(T)$ of affinity indices of the elements of T. The CHOOSE procedure selects an element f with the highest affinity index from T.

Algorithm CLUSTER(G,var T)
Input G(V,E,p): affinity graph;
Output T: set of clusters V;
Step 1:Sort on E to generate a list M of subsets E ($M_1,\ldots,M_k$) such that all edges in the same M_i have the same value of p(e), and if $e_i \in M_i$, $e_i \in M_i$,i<j then $p(e_i)>p(e_i)$;
Step 2:For I=1 to k begin

Let $G_i = \left(V, \bigcup_{j=1}^{i} M_j\right)$;

Find all connected components of G_i;

End;
Step 3:Let T_1 be the set of the vertex sets of all elements with at least 2 vertices generated at Step 2;
Let T_2 consist of the isolated vertices;
$T=T_1 \cup T_2$;
End

Algo 3: Cluster algorithm needed for NOV

For the affinity indices in the CHOOSE procedure of Algo 2, the following defintions are used. Let $G=(V,E,p)$ and Let V' a cluster of G. If $|V'|>1$ then let $E(V')$={e: e is incident to two vertices of V' and $p(e)$ is larger than affinity values of the edges going out from V'} or $E(V')=\phi$. The quadruple $\alpha=(c(V'), x(V'), d(V'), w(V'))$ denotes an affinity index of V' where $c(V')$=2 if the graph $G(V')$ is 2-connected otherwise $c(V')$=1. $x(V')=\frac{1}{|V'|}$. $d(V')=\frac{2|E(V')|}{|V'|(|V'|-1)}$. $w(V')=\frac{\sum_{e\in E(V')} p(e)}{|E(V')|}$ and if $|V'|=1$ then $c(V')$=1, $x(V')$=0, $d(V')$=0, and $w(V')$=0. From these, two quadruples $a_1>a_2$ if **(a)** $c_1>c_2$ or **(b)** $c_1=c_2$ and $x_1>x_2$ or **(c)** $c_1=c_2$ and $x_1=x_2$ and $d_1>d_2$ or **(d)** $c_1=c_2$ and $x_1=x_2$ and $d_1=d_2$ and $w_1=w_2$. Further in [18], an overlapping partitioning (OVP) algorithm is developed using as input the set of fragment $F=\{f_i : 1\le i\le l\}$ where l is the number of fragments from NOV (Algo 2). The idea is to extend the fragments f_i by adding some attributes v as follows: the summation of the transactions T_1 accessing v is greater than the summation of transactions T_2 updating v.

In [12], the algorithm solves the vertical partitioning problem on a different scale. Previously and according to [8,9,18,19,22,23,25], the AAM is the important factor for

solving the vertical partitioning problem. However, because affinity shows how strong a bond is among two attributes, restrictions exist when more than two attributes is the issue. So the calculation of the AAM fails. The optimal binary partitioning of [12] uses instead the AUM that not only solves the affinity problem but also it uses the transaction notion, an important aspects in DDB which was almost missing in the AAM. The purpose of a vertical partitioning is to minimize the total number of disk accesses of equation (9) by partitioning relations into several fragments.

$$\text{Minimize total costs} = \sum_{i=1}^{n} \sum_{j=1}^{d} f_i \left(l(I + F_j) \right) \qquad (9)$$

Such that *n is* number of transactions, *d is* number of fragments generated by the partitioning algorithm. I is primary key (or tuple identifier), $l(I+F_j)$ is the sum of the length of the fragment of F_j and I, and f_i ($l(I+F_j)$) is the cost of accessing a given fragment F_j by a transaction i. Next, the notion of reasonable and unreasonable cuts that are a fundamental concept in this vertical partitioning, are defined. A self-contained fragment is the set of attributes that a transaction accesses. A contained fragment is the union of such self-contained fragments. A reasonable cut divides a relation to two parts (binary) having one of the fragments as being a contained fragment. Further, all binary cuts that are not reasonable are called unreasonable cuts. In a relation of n transactions, there are $2^n - 1$ possible reasonable cuts grouped according to $\binom{n}{i}$ for i=1…n. In reality, both reasonable and unreasonable cuts exist. As the algorithm of [12] works only in a space having only reasonable cuts, a proved theorem (Theorem 1, p. 805 [12]) states that for any unreasonable cut there is a reasonable one that costs less or equal to the unreasonable cut. From theorem 1, the optimal binary partitioning based on the branch and bound algorithm can be implemented (Algo 4). The search tree is constructed as follows. Each node of the tree represents a

transaction in the AUM. The left branch represents the attributes accessed by the transaction that are included in the reasonable cut. The right branch represents the remaining attributes accessed by the transaction that are excluded. The terminal nodes are expanded as more

```
Algorithm OBP(F,F1,F2)
1.      Mincost=evaluate_cost(F); better= 'no better';E-node=root.
2.      Construct the pool of unassigned transactions for the path
3.      If the pool is empty goto 6;
        Select a transaction from the pool and expand the E-node.
4.      Explore the left-branch
        If the lower bound cost including all the attributes accessed by this transaction in the T-
        fragment<mincost then let the terminal node of the left branch be the new E-node and
        goto 2;
   5.   Terminate this branch and backtrack to explore the right branch. Let the terminal
        node of the right branch be the new E-node. Goto 2;
   6.   TempF1=T-fragment; tempF2=F-tempF1;
        currentcost=evaluate_cost(tempF1)+evaluate(tempF2); if currentcost>=mincost
        then goto 7; F1=tempF1;F2=tempF2;mincost=currentcost;better= 'better';
   7.   Let the new E-node be the terminal node of the right branch of the closest ancestor
        from the E-node with a right branch that has not been explored and goto 2;
        If no such ancestor exists then return(better).
```

transactions are considered, thus the tree grows as the processing of the algorithm progresses.

Algo 4: Optimal vertical binary partitioning

Where the *E-node* is the terminal node currently being expanded. The *Path* is the arc connected from the root to the *E-node*. The *T-fragment* is the union of all the attributes accessed by the transactions included in the reasonable cut on the path.

Each of the mentioned vertical algorithms has its specific objective function or way to be implemented, therefore they cannot be compared. In [8,9,22], A common objective function entitled the partition evaluator (PE) is developed. The purpose is to find common aspects among the different algorithms for comparison. Recall from [8,9,18,19,22,23,25], the AAM finds only a bond among a pair of attributes. This means the AAM is limited when the issue is to consider more than two. For this reason, the PE is based on the AUM and not AAM. Another issue helped in the development of the PE is as follows. Local fragments hold the attributes required by transactions, but this is not always the case. These fragments

might also include unneeded attributes by the same transactions but needed by others. Furthermore, some required attributes by the same transactions are stored remotely at other sites. The objective of vertical partitioning is to maximize local accessibility and minimize data access at remote sites. Therefore, the PE that measures the goodness of a vertical partitioning scheme has two corresponding terms (refer to equation (10)).

$$PE = E_L^2 + E_R^2 \tag{10}$$

Where E_L is the "irrelevant local attribute access cost" and E_R is the "relevant remote attribute access cost". Equation (11) shows how to calculate E_L and equation (12) is for E_R.

$$E_L^2 = \sum_{i=1}^{M} \sum_{t=1}^{T} \left(q_t^2 \times |S_{it}| \times \left(1 - \frac{|S_{it}|}{n_i} \right) \right) \tag{11}$$

$$E_R^2 = \sum_{t=1}^{T} \Delta_{i=1}^{M} \sum_{k \neq i} \left[q_t^2 \times |S_{itk}| \times \left(1 - \frac{|S_{itk}|}{n_{itk}^{rem}} \right) \right] \tag{12}$$

where Δ is an operator that computes either the average, the minimum, or the maximum relevant remote attribute cost over all i. T is the total number of transactions that are under consideration. q_t represents the frequency of transactions t for t=1, 2, 3, …, T. M is for the total number of fragments of a partition. n_i is the number of attributes in fragment i. n_{itk}^{rem} is the total number of attributes that are in fragment k accessed remotely with respect to fragment i by t. S_{it} is the set of attributes contained in fragment i that transaction t access. $|S_{it}|$ is the number of attributes contained in fragment i that transaction t access. R_{itk} is the set of relevant attributes contained in fragment k accessed remotely with respect to fragment i by transaction t. $|R_{itk}|$ is number of relevant attributes contained in fragment k accessed remotely with respect to fragment i by transaction t. PE is tested in different manners. The example in this article uses the exhausted enumeration. The minimum number of fragments is 1 and the maximum is n (where n is the number of attributes). Having AUM, PE is

calculated for fragments varying from 1 to n. Then the optimal number of fragments is the one having the minimum PE value. The authors also implemented the PE objective function on BVP [23], and graphical algorithm [25]. An example of 20 attributes and 15 transactions used in BVP [23] and graphical [25] algorithms resulted into 4 fragments and 5 fragments respective. The PE reveals that 4-fragment result is more optimal than 5-fragment result. This example highlights the usefulness of the PE to evaluate the results of the different partitioning algorithms and select the appropriate partitioning algorithm [8,9,22].

4.3 SUMMARY

Figure 12 shows the phases needed to be achieved before implementing one or more of the mentioned vertical techniques.

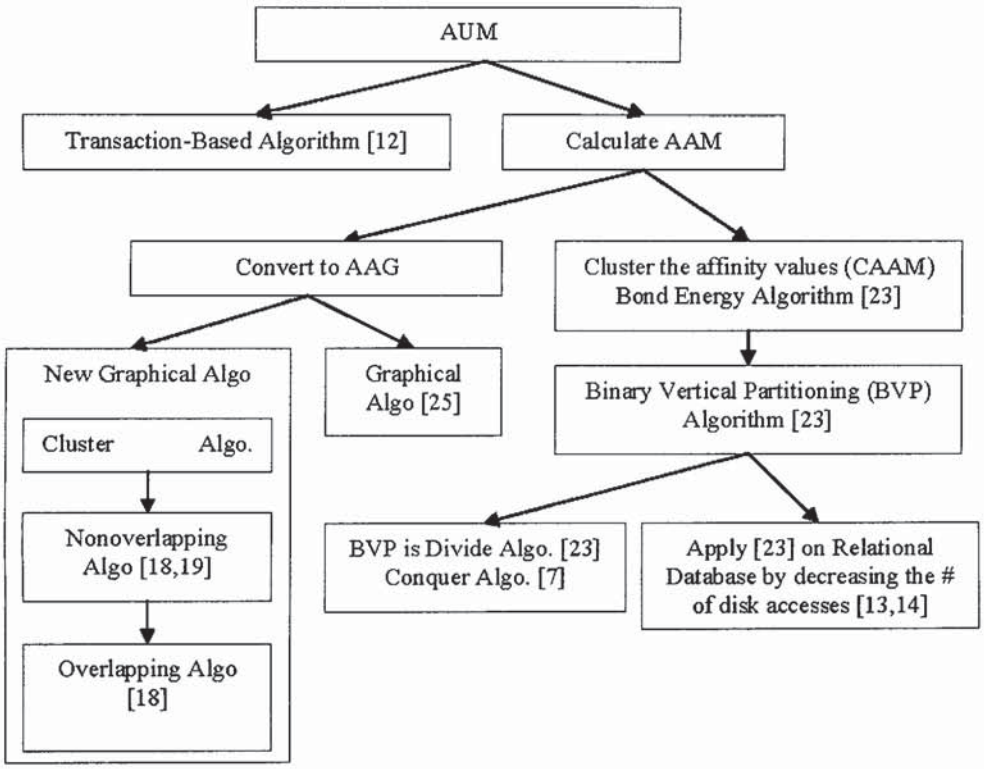

Figure 12: Methodology for the vertical fragmentation algorithms

And Table 9 lists the criteria of each reviewed algorithm.

Algorithm	Input Parameter	Complexity	Need an objective function.	# of iteration needed	# of fragments after 1st iteration.	Real system tested	Disk accesses	Creating overlapping fragments	Reasonable fragments
BVP [23]	CAAM	$O(n^2 \log n)$	Yes	Many	2	Yes	N/C	Yes	Yes
Graph [25]	AAG	$O(n^2)$	No	1	All	No	N/C	No	Near-optimal
Conquer*[7]	CAAM	$O(n^2)$ to $O(n^4)$	Yes	Many	2	Yes	Optimize	Yes	Yes
Trans-Based [12]	AUM	$O(2^m)$	No	Many	2	Yes	Optimize	No	Yes
NOV[18,19]	AAG	$O(ldn)$	No	1	All	Yes	N/C	No	Yes
OVP [18]	NOP	$O(ldn + mn^2)$	No	1	All	Yes	N/C	Yes	Yes

Table 9: Comparison table of the vertical algorithms

Note:
CAAM a Clustered AAM that is calculated using AUM. CAAM increased the complexity of AAM by an order of n.
For vertical partitioning algorithms that use attribute affinity, the complexity of AAM is on the order of $O(n^n)$[23].
AAG is the interpretation of the AAM to a graph.
* The conquer tool does not change the vertical partitioning module of [23] in Figure 7 but it considers additional modules and better evaluation. This increases the complexity of the system.

N/C : Not taken into consideration
m : Number of transactions in the system
n : Number of attributes in the system
d : Number of edges in the AAG
l : Number of fragments is of $O(n)$

CHAPTER V- THE HORIZONTAL FRAGMENTATION TECHNIQUES

The horizontal fragmentation is another form to partition a relation. This chapter focuses on different algorithms to solve this type of fragmentation. It begins with the identification of the horizontal partitioning. Next an overview of different algorithms for solving this issue is suggested. The chapter is concluded with a summary that I have grouped showing the relationships among different algorithms and another summary showing the most important criteria in the studied routines.

5.1 Definition

Horizontal fragmentation consists of selecting subsets of tuples from the global relation according to some predicates. Each subset forms a fragment. Thus, each fragment has the same schema as the global relation. A predicate is used as the basis of any horizontal partitioning. It is a boolean function made up of clause <domain><oprator><value>. The <domain> is the attribute domain on which the condition is applied, <operator> may be one of the elements in the set $\{=, \neq, <=, <, >=, >\}$ and the <value> is any admissible value for the domain [15,16,20,21,28].

5.2 Overview

In [5], horizontal fragmentation is achieved by creating minterm fragments. A minterm is the conjunction of all simple predicates taken either in the normal or negated form, i.e., the set $Y(X)$ of minterm predicates $y_1,...,y_m$ associated to set X of simple predicates is defined in equation (13).

$$y_i = \bigwedge_{x_j \in X} x_j^* / x_j^* = x_j \text{ or } x_j^* = \neg x_j \Leftrightarrow y_i = x_1^* \wedge x_2^* \wedge ... \wedge x_n^* \qquad \textbf{(13)}$$

In other words,

$$y_1 = (x_1 \wedge (\neg x_1)) \wedge (x_2 \wedge (\neg x_2)) \wedge \ldots \wedge (x_n \wedge (\neg x_n))$$
$$y_2 = (x_1 \wedge (\neg x_1)) \wedge (x_2 \wedge (\neg x_2)) \wedge \ldots \wedge (x_n \wedge (\neg x_n))$$
.
.
$$y_m = (x_1 \wedge (\neg x_1)) \wedge (x_2 \wedge (\neg x_2)) \wedge \ldots \wedge (x_n \wedge (\neg x_n))$$

y_i is a minterm. Obviously the number of minterms $m = 2^n$, having n the number of simple predicate. A minterm fragment is defined as a subset of records of file F such that y_i is true. The designer before using this type of fragmentation must be aware of the allocated sites where the fragments should be found.

In [4], an optimization model is developed for the horizontal partitioning using the DDB design methodology of Figure 6. The user is required to specify the information about **(a)** the data (attribute size, relationship size, cardinality), **(b)** the tabulation of transactions (frequencies, site of origin), and **(c)** the distribution requirement as mentioned in Figure 6 on page 13. Two types of horizontal fragmentation are identified. The first is the primary partitioning where the fragmentation is directly applied on the relations and the second is the derived partitioning where the fragmentation is indirectly applied on relations via one or more links. An objective function (equation (14)) in the form of a linear integer zero-one programming is developed for the nonreplicated distributed database based on the transaction given occurrences. The function to be minimized is:

$$\min z = \sum_{i,p} C_{ip} X_{ip} + \sum_{i,j} D_{ij} Y_{ij} - \sum_{h,p} A_{hp} W_{hp} - \sum_{h,j} B_h V_{hj} \qquad (14)$$

having C the cost of transaction related to the decision of partitioning object i according to partition p. D is the cost of transaction related to the decision of allocation object i according to partition p. A is the cost of transmissions that can be saved because of the use of the same partitioning criteria p on the owner and the member of the link h. B is the cost of transmissions that can be saved because both the owner and the member of the link h are

stored on the same site j as a whole. X=1 if candidate partitioning p is selected for object i, else =0. Y=1 if the object i is allocated on the site j as a whole, =0 otherwise. W=1 if link h is used for deriving a partition in the hierarchy of partitioning p, =0 otherwise. V=1 if link h is local to site j, =0 otherwise.

In [29], A horizontal graph-based partitioning is adopted in order to find all meaningful horizontal fragments. The process begins by selecting the transactions having high number of occurrences. Each transaction is using one or more predicates. These transactions and their used respective predicates are grouped in a matrix form known as the predicate usage matrix (PUM). The structure of the matrix is the same as the AUM. Meaning that the transactions are written at the row level and the predicates at the column level. The elements of the PUM are 1 if the transaction T_i is accessing a predicate P_j otherwise it is 0 (Table 10). An additional column is added to the ones of predicates showing the number of accesses of each transaction. As an example, consider the following transactions on a relation including the attributes D#, SAL and AGE among others:

```
T1: D#<10 (P1), SAL>40000 (P7)
T2: D#<20 (P2), SAL>40000 (P7)
T3: D#>20 (P3), SAL>40000 (P7), AGE<30 (P9)
T4: 30<D#<50 (P4), SAL<40000 (P8), AGE<30
T5: D#<15 (P5), SAL<40000 (P8)
T6: D#>40 (P6), SAL<40000, AGE>30 (P10)
T7: D#<15 (P5), SAL<40000 (P7)
T8: D#<10 (P6), SAL<40000, AGE>30
```

The PUM of the previous example is:

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	Access
T1	1	0	0	0	0	0	1	0	0	0	25
T2	0	1	0	0	0	0	1	0	0	0	50
T3	0	0	1	0	0	0	1	0	1	0	25
T4	0	0	0	1	0	0	0	1	1	0	15
T5	0	0	0	0	1	0	0	1	0	0	25
T6	0	0	0	0	0	1	0	1	0	1	25
T7	0	0	0	0	1	0	0	1	0	0	25
T8	0	0	0	0	0	1	0	1	0	1	15

Table 10: A predicate usage matrix

Next, the predicate affinity matrix (PAM) is calculated using the same equation (1) as to calculate the AAM on page 17. The difference is that acc_k represents the number of accesses of transaction k accessing both predicates i and j. Thus, the resultant PAM of Table 10 will be in Table 11.

	1	2	3	4	5	6	7	8	9	10
1										
2	<== *									
3	*	*								
4	0	0	==>				*Symmetric*			
5	0	0	0	*						
6	0	0	0	*	*					
7	25	50	25	0	0	0				
8	0	0	0	15	50	40	*			
9	0	0	25	15	0	0	25	15		
10	0	0	0	0	0	40	0	40	0	

Table 11: A predicate affinity matrix of Table 10 with 2 symbols

However, the PAM is a symmetric matrix that includes two additional notations '==>' and '*'. The value '==>' of the element (i,j) indicates that predicates i implies predicate j. The value '*' represents that predicates i and j are in a way close together although they do not necessary implies implication. Two predicates are close if **(a)** i and j must be defined on the same attribute, **(b)** i and j must be used jointly with some common predicate h and **(c)** predicate h is defined using another attribute than the one used of i and j. For example, *P2* and *P3* in the previous example are close. Next, the Algo 1 is implemented with a slight modification as shown in Algo 5. The changes are done in way to take into account the two additional notations using the following heuristic rules **(a)** Any non-zero value have higher priority than '*' and '==>' when selecting the next edge. **(b)** When checking for a possible edge, the edges with these notations are ignored. **(c)** The value '==>' has a higher priority than '*'. These rules can be easily incorporated in Algo 1 if we consider all affinity values greater than 2 because 1 and 2 are assigned to '*' and '==>' respectively. Obviously, since the algorithm considers only the important transactions with high

occurrences, the affinity value must be greater than 2. Thus, the Algo 1 is modified

Construct the predicate usage matrix.
Construct the predicate affinity matrix. It includes two additional symbols the '*' and '→'.
Construct the predicate affinity graph.
From the affinity graph, construct the spanning tree:
Begin with any node.
Select an edge having the highest value. Note: any value is considered as high then '*' and '→'.
When checking for a cycle the value '*', '→' are ignored.
The value of → is considered of having a high value as '*'.
A set of clustered is determined. Optimize the clustered predicates.
Compose predicate terms using predicate term schematic table.
Place in the 1st column the chosen attribute with the appropriate ranges to cover that attribute exhaustively
Place in the 2nd column the next least common attribute and write its appropriate ranges that appear in the cluster sets against the 1st column
If the considering least common attribute is opened to the attribute that is in the left column, the considering predicate must be written as many times as the number of predicates in the left column

providing a horizontal fragments to Algo 5.

Algo 5: Horizontal graphical fragmentation

The optimization process in step 5 of Algo 5 is done just to reduce the number of predicates. If necessary the set of clusters of Table 11 is shown in Figure 13.

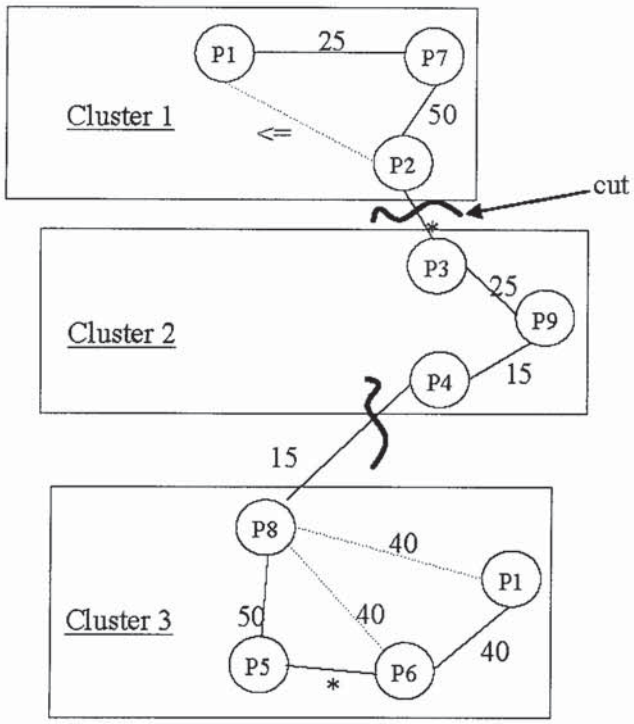

Figure 13: Clustering of predicates of Table 11

The 3 clusters of Figure 13 are: cluster 1={D#<20,SAL>40000}, cluster 2={D#>20,AGE<30} and cluster 3={D#<15,D#>40,SAL<40000,AGE>30}. The schematic table of these three clusters is illustrated in Table 12

SAL>40000 (P7)	AGE <30 (P9)	D#<20 (P1,P2)
		D#>20 (P3,P4)
SAL<40000 (P8)	AGE>30 (P10)	D#<15 or D#>40 (P5) (P6)
	AGE<30 (P9)	D#>30 (P3,P4)
ELSE		

Table 12: A predicate term schematic table

Each predicate term represents a fragment, i.e., Table 12 shows five different fragments including {SAL>40000 AND AGE<30 AND D#<20} and {SAL<40000 AND AGE>30 AND (D#<15 OR D#>40)}, and others. The ELSE fragments in Table 12 is added to insure the completeness of the horizontal fragmentation.

In [31], a similar algorithm to the BVP of [23] is developed and implemented to obtain horizontal fragments. The PAM is calculated through the PUM using the same

equation (1) as in [29]. But the PAM does not include the two symbols '*' and '==>'. Thus the PAM of Table 11 of the same example becomes Table 13.

	1	2	3	4	5	6	7	8	9	10
1	25	0	0	0	0	0	25	0	0	0
2	0	50	0	0	0	0	50	0	0	0
3	0	0	25	0	0	0	25	0	25	0
4	0	0	0	15	0	0	0	15	15	0
5	0	0	0	0	50	0	0	50	0	0
6	0	0	0	0	0	40	0	40	0	40
7	25	50	25	0	0	0	100	0	25	0
8	0	0	0	15	50	40	0	105	15	40
9	0	0	25	15	0	0	25	15	40	0
10	0	0	0	0	0	40	0	40	0	40

Table 13: A predicate affinity matrix of Table 10 without 2 symbols

Next, the PAM is clustered in the same manner of CAAM creating the clustered predicated affinity matrix (CPAM). Same as previously, two cases pops up. For the nonoverlapping, a single point x is placed on the on the CPAM such that the equation (1) is maximized. The parameters *CL*, *CU* and *CI* are calculated as previously on page 19. For the overlapping the same objective function is maximized but the parameters *CL*, *CU* and *CI* are calculated taking into consideration the update and retrieval transactions of Table 5. Algo 6 shows the different steps needed for the development of the binary horizontal partitioning algorithm.

Input a PUM
Construct PAM
Transform PAM to CPAM
Optimize the predicates (using the inclusion and refinements) to minimize the number of predicates
Construct horizontal fragment using either

Algo 6: Binary horizontal partitioning

5.3 SUMMARY

Figure 14 of this chapter shows the phases needed to be achieved before implementing one or more of the mentioned vertical techniques.

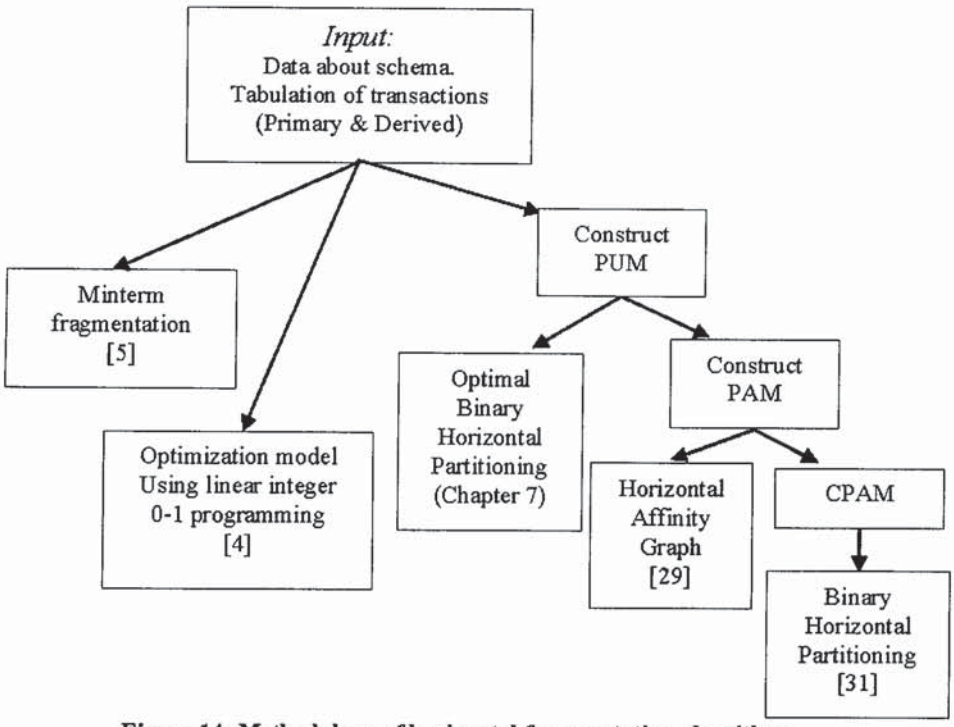

Figure 14: Methodology of horizontal fragmentation algorithms

Optimal binary horizontal partitioning is a proposed horizontal fragmentation based on the PUM because affinity value has a deficiency, it finds the bond only among two predicates. Thus calculating affinity value for more than two predicates fails. To solve this problem, a new horizontal algorithm is proposed in chapter 7 using the vertical partitioning algorithm of [12].

Table 14 lists the important criteria in the studied algorithms

Algorithm	Input Parameter	Complexity	Need an objective function.	# of iteration needed	# of fragments after 1st iteration.	Real system tested	Disk accesses	Creating overlapping fragments	Reasonable fragment
Minterm [5,28]	predicate	$O(n^2)$	No	1	At most 2^n	Yes	N/C	Yes	Yes
Graph [29]	PUM	$O(n^2)$	No	1	All	Yes	N/C	No	Yes
BHP [31]	CPAM	$O(n^2 \log n)$	Yes	Many	2	Yes	N/C	No	Yes
OBP (Ch. 7)	PUM	$O(2^m)$	No	Many	2	No	Yes	No	Yes

Table 14: Comparison table for the horizontal algorithms

N/C : Not taken into consideration
n : Number of predicates
m : Number of transactions

Note: The Graph and BHP are based on the predicate affinity matrix (PAM) which has a complexity on the order of $O(n^n)$

In case where the vertical fragmentation is implemented on the horizontal fragments, the result will be a mixed fragmentation, a third way of fragmenting a relation. Thus, creating mixed fragments is an easy task when compared to check the completeness and reconstruction features of the DDB fragmentation. This chapter shows the steps required achieving a mixed partitioning. Next and because these steps are relatively an easy task to used, the focus will be on how to maintain the two features of DDB fragmentation, reconstruction and completeness in a mixed fragmentation.

6.1 Definition

A mixed fragmentation is a combination of both horizontal and vertical partitioning. A mixed fragment can be achieved in two steps. First, sets of tuples are selected from the relation upon a given predicate. Next, the resulting relations are projected onto a set of attributes. In other words, a mixed fragmentation is a combination of a selection and a projection or vice versa [15,16,20,21,24]

6.2 Overview

Any vertical algorithm mentioned in chapter 4 combined with one of the horizontal algorithm of chapter 5, results in the formation of mixed fragments. In [24], the graphical vertical algorithm (Algo 1) and the graphical horizontal algorithm (Algo 5) are combined to find all the meaningful mixed fragments. The resulting combination is $n \times m$ sets of grid cells where n is the number of vertical fragments and m is the number of horizontal fragments. Mathematically, the vertical fragments of the horizontal partitioning represented by $(1_p, 2_p, ..., n_p)$ form horizontal grid cells. The horizontal fragment of a vertical partitioning represented by $(i_a, i_b, ..., i_m)$ for vertical grid cells (Figure 15). However, considering each grid cell as a fragment is inconvenient because a transaction may always

access a certain number of grid cells repetitively. Thus the number of joins increases among these cells. For this reason, the authors of [24] have merged these grids forming a regular fragment. Next, they have proved that a transaction accesses only a regular fragment. To reach this theorem including the explanation of a regular fragment, a set of criteria and

definitions are established as follows.

Figure 15: Representation of grid cells

Using the two representations $(1_p,2_p,...,n_p)$ and $(i_a,i_b,...,i_m)$, two operations concatenated $(\|)$ the horizontal merging operator and union $(\cup)$ the vertical merging operator are defined on the set of horizontal and vertical grid cells. Let two vertical grid cells i_p and i_q the union is $(i_{p,q})=i_p \cup i_q$. Let two horizontal grid cells i_p and j_p the concatenation is $(i_p,j_p)=i_p\|j_p$. The binary operations union and concatenation are commutative and associative over the set of vertical and horizontal cells respectively. Mathematically, a grid cell is represented as α_β where $\alpha=1,2,...,n$ the column index and $\beta=a,b,...,m$ the row index.

Def. 1: A well formed expression *w* is defined as follows:

1. $w=\alpha_\beta$ is well formed expression represented by $r(w)=\alpha_\beta$, where $\alpha=1,2,...,n$ and $\beta=a,b,...,m$.

2. $w = \alpha_{1\beta_1} \| \alpha_{2\beta_2}$ is well formed if $\alpha_1 \neq \alpha_2$ and $\beta_1 = \beta_2$. $r(w) = (\alpha_{1\beta_1}, \alpha_{2\beta_2})$ is its representative. Also, $w = \alpha_{1\beta_1} \bigcup \alpha_{2\beta_2}$ is well formed if $\alpha_1 = \alpha_2$ and $\beta_1 \neq \beta_2$. $r(w) = (\alpha_{1(\beta_1\beta_2)})$ is its representative.

3. $w' = \alpha_\beta \| w$ is well formed if there exist a grid cell represented by α'_β in $r(w)$. $r(w') = r(w) \bigcup \alpha_\beta$ is its representative.

4. Any number of invocations of the above set of rules.

In other words, in forming a well formed expression, two cells are concatenated only if they are both horizontal grid cells from the same horizontal fragment and two cells are unified only if they are both of vertical grid cells of the same vertical fragment. In general a well formed expression w is represented as $(\alpha_{1A_1}, \alpha_{2A_2}, ..., \alpha_{pA_p})$ where α_i represent a vertical fragment and A_i represent a set of vertical grid cells.

Def. 2: A regular well formed expression is a well formed expression

$$w = \left\| \begin{matrix} i=p \\ i=1 \end{matrix} \left\{ \bigcup_{\beta \in A_i} (\alpha_i \beta) \right\} \qquad A_1 = A_2 = ... = A_p = A \qquad \textbf{(15)}$$

in other words the set of vertical grids cells A_i in the formation of the fragment is the result of a regular well formed expression over the set of grid cells.

Def. 3: A fragment is the result of a well-formed expression over a set of grid cells. A regular fragment is the result of a regular-formed expression over the set of grid cells.

For example, in Figure 15, 1_a, 1_b, 2_a, 2_b form a regular fragment by merging the respective grid cells, where as 1_a, 1_b, 2_a are not regular. In this case null values are used to fill up the gaps. Remember that a regular fragment is a table in the relational database.

Furthermore, a transaction projects a number of attributes to a relation and selects tuples upon a predicate thus creating a mixed fragment. Using the above information and

definitions, the authors in [24] proved that transactions only access regular fragments as it has been mentioned earlier.

In [21], the authors do not suggest how to create a mixed fragmentation because it is an easy task but they emphasize on how to preserve the two features: completeness and reconstruction. Thus, an algorithm is developed to check the completeness of a fragmentation schema. (It is worth to mention that fragmentation means a mixed fragmentation. However, this meaning does not exclude the vertical and the horizontal partitionings, as they are considered special cases of a mixed fragmentation). Completeness is defined: "all data are mapped onto some fragment so that no item is lost" [21]. And reconstruction "requires that the global schema can be reformed from its derived fragments"[21]. It is obvious from the definitions that if the completeness feature is satisfied so is the reconstruction. Thus, the authors suggested an algorithm (Algo 7) to check the completeness of a fragmentation design. However some preliminary background and notations are required.

Completeness means no data loss. The type of data can be a complete tuple in case of horizontal fragmentation. It can be an attribute and its domain in case of vertical partitioning. Further the data can be either one of the mention partitioning or a combination of both fragmentation, i.e., an item in case of mixed fragmentation. Checking the completeness of either horizontal or vertical is straightforward. But it is difficult to achieve the request in the mixed fragmentation. Mathematically, completeness can be defined as:

Def. 4: Given a fragmentation schema F, a fragmentation attribute is any attribute of $s(r)$ on which a simple predicate of a qualification is defined. The fragmentation set $fs(r)$ is the ordered set of the fragmentation attributes. The fragmentation domain $C(r)$

is the cartesian product of the domains of the fragmentation attributes, taken in the order defined by $fs(r)$ (equation (16)).

$$C(r)= \prod_{A_i \in fs(r)} \mathrm{dom}(A_i) \qquad \textbf{(16)}$$

for any tuple t of $C(r)$, the image of t is the set of fragments whose qualification is satisfied by t: $i(t)= \{f_j / p_j(t)\}$ p: partition

Def. 5: A set of fragments $F = \{f_1, f_2, \ldots, f_k\}$ is said to cover a list of attributes X if the union of the attribute lists of the fragment in F is a superset X: $X \subseteq \bigcup_{f_i \in F} L_i$, from the above definition it is easy to states the condition of completeness.

Postulate 1: the fragmentation schema of a global relation r is complete if and only if the image of any tuple $C(r)$ covers the relation schema $s(r)$.

Three types of violations may occur to this postulate. **(a)** If for a tuple t in $C(r)$, $i(t)$ is empty then there is a tuple loss for any tuple t of r whose fragmentation attributes values are those in t. **(b)** If $i(t)$ does not cover $s(r)$ but is not empty then there is an item loss, in correspondence with the attributes not covered by $i(t)$. And **(c)** If the same attribute is missing in $i(t)$ for all $t \in C(r)$ then there is an attribute loss. Algo 7 is an efficient algorithm to check the completeness of a mixed fragmentation.

Input: A fragmentation schema F
Output: "Yes" if F is complete; otherwise a list of the tuples of $C(r)$ that violate completeness, with the associated image.

Begin

Set the violation list empty

For each tuple t of $C(r)$, compute $i(t)$ and check whether $i(t)$ covers $s(r)$; if not add the pair $(t, i(t))$ to the violation list;

If the violation list is empty, then output "yes"; otherwise output the violation list.

Algo 7: Checking the completeness of a mixed fragmentation schema

6.3 Summary

In this chapter we have seen that the mixed fragmentation is achieved by applying the vertical partitioning on the horizontal fragments or vice versa. Thus making the mixed fragmentation task easier then the previous ones. Therefore and as we have reviewed, the importance of such partitioning is how to preserve its completeness because while creating mixed fragments, data may be lost.

However, an interesting point is to check which vertical algorithm to use with which horizontal one to give an optimal number of mixed fragments.

CHAPTER VII- A PROPOSED HORIZONTAL FRAGMENTATION ALGORITHM

7.1 Introduction

Most of the horizontal partitioning algorithms namely the graphical algorithm and the binary horizontal are based on the PAM. However the PAM by itself has its own deficiencies. First, if finds the affinity value among two predicates only. The calculation of PAM fails when the objective is to find a bond among more than two predicates. Second, PAM lacks the semantic meaning of transactions. Therefore, the two algorithms have used as input the predicate only. Because the transactions have the same importance as the predicates, another horizontal algorithm is proposed in this chapter taking into consideration both transactions and predicates into account as input. This means the input of the algorithm is not anymore the PAM but the PUM. When considering the PUM, the complexity of the problem is reduced from $O(n^n)$ where n is the number of predicates (complexity of the PAM without forgetting the complexity of the fragmentation algorithm used) to $O(2^m)$ having m the number of transactions.

7.2 Solving the Proposal

Using the concept of minimizing the total number of disk accesses of page 29, things are quite different in case of horizontal partitioning, such as a transaction accesses a set of tuples as a whole as stated in the definition. Because the horizontal fragments have the same schema as their derived global relations, the length of the tuple is always constant. Therefore the cost equation (9) on page 29 will become irrelevant and thus, the space area cannot be determined. However, to solve the horizontal problem and to be able to implement the algorithm of [12], the equation 2 must be altered to equation (17).

$$\text{minimize total cost} = \sum_{i=1}^{n} \sum_{j=1}^{d} f_i(c_j) \qquad (17)$$

where c_j is the cardinality of fragment j, $f_i(c_j)$ is the cost of accessing c_j tuples of fragment j by transaction i, n is the number of transactions and d is the number of fragments generated by the partitioning algorithm. As a consequence, the theorem 1 on page 29 must be proved using the cost equation (17). Remember that the algorithm of [12] works on reasonable cut space area where the unreasonable cuts were reduced by the mentioned theorem.

Similarly to [12], *Theorem 2*: let c_j be the cardinality of fragment j and let $f_i(c_j)$ be the cost of accessing c_j tuples of fragment j by transaction i. If the second derivative of $f_i(c_j)$, $f_i''(c_j)<0$ (concave downward) for all i, then for a given unreasonable cut of a relation, there exists at least one reasonable cut that yields less or equal than that of the unreasonable cut.

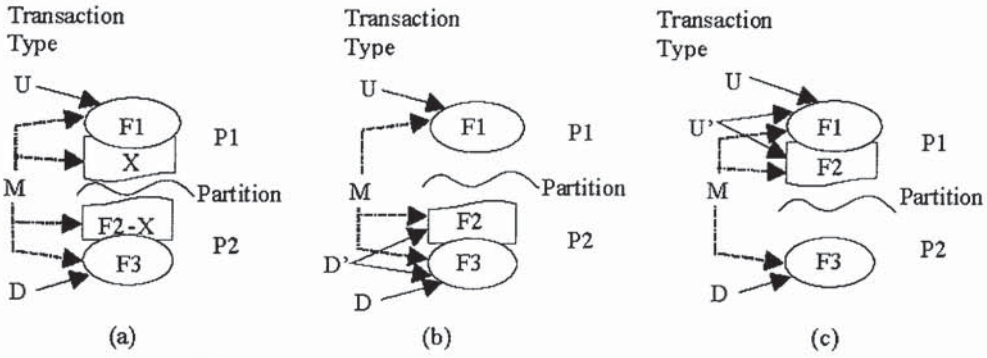

Figure 16: Reasonable and unreasonable cuts of a relation

Proof. An unreasonable cut divides a relation in two parts as shown in Figure 16(a). The tuples are subdivided into *P1* and *P2*. Let types *U* (Up) and *D* (Down) transactions access the tuples only in *P1* and *P2* respectively. *F1* is the set of tuples accessed by *U* and *F3* is the set of tuples accessed by *D*. Let *X* be the set of tuples not accessed by *U* transactions in *P1* and *F2-X* is the one not accessed by *D* transactions in *P2*. Type *M* (Middle) transactions access both partitions *P1* and *P2*. Consider the Figure 16(b) where the cut is moved to the boundary of *F1*. The cost (accessing a fragment) of type *U* transactions is not affected. But the cost of type *M* transactions may decrease since some tuples are moved

from *P1* to *P2*. Those type *M* transactions that only access *X* no longer need to access tuples in *P1* after the partition is moved to the boundary *F1*. Let D' be this type of transactions. Similar to U' in Figure 16(c) where U' represents the type *M* of transactions that only access *F2-X* which does no longer access *P2* after the partition has been moved to the boundary of *F3* fragment.

Let C_u be the cost of the unreasonable cut. Let C_{r_1} be the cost of the reasonable cut of Figure 16(b) (partition moved to the boundary of *F1*) and let C_{r_2} be the cost of the reasonable cut of Figure 16(c) (partition moved to the boundary of *F3*). Thus, from equation 17, we have:

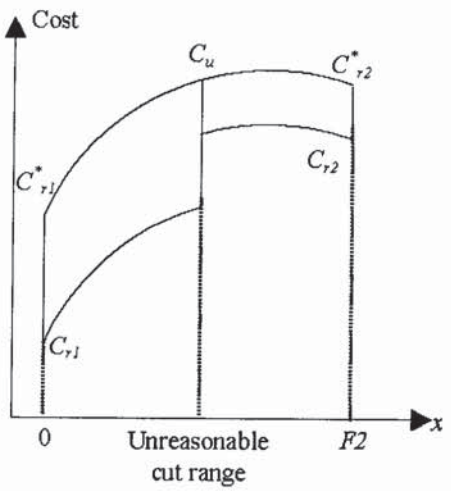

Figure 17: Cost region for unreasonable cut

$$C_u(x) = \sum_{i \in U \cup M} f_i(c_1 + x) + \sum_{i \in M \cup D} f_i(c_3 + (c_2 - x)) \qquad \textbf{(18)}$$

such that $x \in [0..F2]$.

$$C_{r_1} = \sum_{i \in U \cup M - U'} f_i(c_1) + \sum_{i \in M \cup D} f_i(c_3 + c_2) \qquad \textbf{(19)}$$

$$C_{r_2} = \sum_{i \in U \cup M} f_i(c_1 + c_2) + \sum_{i \in M - D' \cup D} f_i(c_3) \qquad \textbf{(20)}$$

Let $C^*_{r_1}$ represents the cost of $C_u(x)$ at the end point 0 of the interval [0..F2] and $C^*_{r_2}$ represent the costs for $C_u(x)$ at the end point F2 of the boundary of [0..F2] (Figure 17).

$$C^*_{r_1} = \sum_{i \in U \cup M} f_i(c_1) + \sum_{i \in M \cup D} f_i(c_3 + c_2) \geq C_{r_1} \qquad \textbf{(21)}$$

$$C^*_{r_2} = \sum_{i \in U \cup M} f_i(c_1 + c_2) + \sum_{i \in M \cup D} f_i(c_3) \geq C_{r_2} \qquad \textbf{(22)}$$

$C^*_{r_1} \geq C_{r_1}$ because the first part of the function cost $C^*_{r_1}$ of equation (21) has the interval $U \cup M$ while the first part of the function cost C_{r_1} of equation (19) has the interval $U \cup M - U'$. This insures that $C^*_{r_1}$ costs more than C_{r_1}. Similarly, for $C^*_{r_2} \geq C_{r_2}$ using the equations (22) and (20) respectively. Furthermore, $C^*_{r_1} - C_{r_1}$ is the reduction in access cost by type U' transactions which do not require the access of *P1* and $C^*_{r_2} - C_{r_2}$ is the reduction in access cost by type D' transactions which do not require the access of *P2* (refer Figure 17). Due to concavity of $f_i(c_j)$, we have:

$$\frac{d^2 C_u(x)}{dx^2} = C''_u(x) = \sum_{i \in U \cup M} f''_i(c_1 + x) + \sum_{i \in M \cup D} f''_i(c_3 + (c_2 - x)) < 0 \qquad \textbf{(23)}$$

Therefore, $C_u(x) \geq \min(C^*_{r_1}, C^*_{r_2})$. As $C^*_{r_1} \geq C_{r_1}$ from equation (21) and $C^*_{r_2} \geq C_{r_2}$ from equation (22) then $C_u(x) \geq \min(C_{r_1}, C_{r_2})$. This result shows that there is a reasonable cut having a cost smaller or equal to the unreasonable cut.

The *theorem 2* facilitates the implementation of OBP for vertical partitioning (page 29) on horizontal partitioning using the predicates instead of attributes. The OBP algorithm Algo 4 is applied exactly the same way as in [12] listed on page 30 with only a slight modification while constructing a search tree. The PUM is used as input for reasonable cuts instead of AUM and the lower bound cost includes all the tuples selected by a predicate accessed by the transaction. Thus the algorithm Algo 4 based on the branch and bound concept becomes Algo 8. Of course the optimal binary horizontal partitioning produce two fragments. To obtain the desired fragments, the same algorithm is implemented recursively on each two sub-fragments separately.

```
Algorithm OBP(F,F1,F2)
1.    Mincost=evaluate_cost(F); better= 'no better';E-node=root.
2.    Construct the pool of unassigned transactions for the path
3.    If the pool is empty goto 6;
      Select a transaction from the pool and expand the E-node.
4.    Explore the left-branch
      If the lower bound cost including all the tuples selected by a predicate accessed by this
      transaction in the T-fragment<mincost then let the terminal node of the left branch be
      the new E-node and goto 2;
5.    Terminate this branch and backtrack to explore the right branch. Let the terminal
      node of the right branch be the new E-node. Goto 2;
6.    TempF1=T-fragment; tempF2=F-tempF1;
      currentcost=evaluate_cost(tempF1)+evaluate(tempF2); if currentcost>=mincost
      then goto 7; F1=tempF1;F2=tempF2;mincost=currentcost;better= 'better';
7.    Let the new E-node be the terminal node of the right branch of the closest ancestor
      from the E-node with a right branch that has not been explored and goto 2;
      If no such ancestor exists then return(better).
```

Algo 8: Proposed horizontal algorithm

7.3 Simulation of the Algorithm

A simulation of the algorithm is developed using the C++ language for DOS. Even though the algorithm Algo 8 is well explained but it can not be implemented as is. Several issues must be taken into consideration. The first task of course is to determine the set of variables needed for the simulation. These variables include the *root*, *Enode* used for the construction of the tree, they have the tree data type; the *Tfrag*, *F1*, *F2*, *TempF1*, *TempF2* are used for the fragmentation process. The latter variables are of type arrays having as size the number of predicates involved in the simulation because in worst case there can be at most *pr* fragments where *pr* is the number of predicates. A stack data type is also to achieve the steps 6 and 7 of Algo 8. The second task, which is the most important one, is to evaluate the cost. The calculation of the cost is used in steps 1, 4 and 6 (Algo 8). Thus, it is an important factor is the mentioned algorithm. In order to be able to continue the simulation, I have to determine the inputs required calculating the cost function. Recall that it is an $f(c)$ (equation (17)) where c is the cardinality of the fragment. The user supplies the number of predicates, the number of transactions and the PUM in a file DATA.TXT. The frequencies of each transactions are randomly generated from 0..*OCCURRENCES* where the user supplies *OCCURRENCES* value in DATA.TXT along with the *CARDINALITY* of the relation.

Furthermore, an additional input in an array data type showing the cardinality of each predicate. I consider that if a transaction uses more than one predicate the number of tuples is the sum of both cardinalities. Thus, the cost function will be the cardinality multiplied by the frequency of the transaction involved. Further, if a predicate is used by, let's say two transactions and they are both assigned thus the frequency is the summation of both frequencies multiplied by the cardinality involved. For example, As a typical DATA.TXT file:

```
30000 8 10 50
1 0 0 0 0 0 1 0 0 0
0 1 0 0 0 0 1 0 0 0
0 0 1 0 0 0 1 0 1 0
0 0 0 1 0 0 0 1 0 0
0 0 0 0 1 0 0 1 0 0
0 0 0 0 0 1 0 1 0 1
0 0 0 0 1 0 0 1 0 0
0 0 0 0 0 1 0 1 0 1
```

where 30000 is the cardinality, 8 is the number of transactions, 10 is the number of predicates, 50 is *OCCURRENCES*. And the '1' and '0' are the elements of the PUM. A complete code of the simulation written in C++ language is provided in the Appendix at the end the thesis. Finally, the simulation is repeated 50 times showing the resulted first two fragments. These results saved RESULT.TXT file are highly dependent on the generated number.

7.4 Summary

In this chapter, we have proved that a vertical fragmentation algorithm using the AUM can be implemented for the horizontal partitioning using of course the PUM instead of the AUM. A further study can be done on the cost evaluation function in order to include additional parameters and thus refining the fragmentation process.

CHAPTER VIII- CONCLUSION AND FUTURE WORK

In this thesis, we have presented the data fragmentation problem in homogeneous DDB that aims to increase the locality of reference and decrease the data remote accesses. We reviewed different algorithms that are suggested to solve the issue of vertical, horizontal or mixed, the three types of database partitioning. I have supplied at the end of each chapter for the first two types, a comparative table showing their criteria of the algorithms involved. Next, I have developed an optimal binary horizontal partitioning with a complexity of $O(2^n)$ where n is number of transactions. Last, I have implemented a simulation of optimal binary partitioning using the C++ language for DOS. However, A further study can be done on the cost evaluation function used in the simulation in order to include additional parameters and thus refining the fragmentation process.

As an optimal binary vertical partitioning [12] and an optimal binary horizontal partitioning algorithms (Chapter 7) are developed, a further study can be continued to see the effect of creating a mixed partitioned fragments and check if their number is also optimal.

Further, as a common objective function [8,9,22] is developed for the vertical fragmentation problem, which evaluates the different vertical techniques, a similar issue can be done for the horizontal. We saw that each horizontal technique has its own objective function or a way to be implemented that cannot be used on any other algorithm, a common horizontal objective function can be developed to evaluate the different horizontal techniques.

The database involved in this thesis is focused on relational type only. As the database concept is evolving rapidly and turns out to involve others types including objects, multimedia, like sound, voice images, animation, etc., studies can go into deeper quest to find solutions to the fragmentation problem for these issues.

REFERENCES

[1] Peter M. G. Apers, "Data Allocation in Distributed Database systems", *ACM Transactions on Database Systems*, vol. 13, no. 3, pp. 263-304, 1988.

[2] J.A. Bakker, "A Semantic Approach to Enforce Correctness of Data Distribution Schemes", *The Computer Journal*, vol. 37, no. 7, pp. 563-575, 1994.

[3] David Bell, Jane Grimson, *Distributed Database Systems*, Addison-Wesley Pub, 1992.

[4] Stefano Ceri, Shamkant Navathe, and Gio Wiederhold, "Distribution Design of Logical Database Schemas", *IEEE Transactions on Software Engineering*, vol. SE-9, no. 4, pp. 487-503, July 1983.

[5] S. Ceri, M. Negri, G. Pelagatti, "Horizontal Data Partitioning in Database Design", *ACM*, pp. 128-136, 1982.

[6] Stefano Ceri, Giuseppe Pelagatti, *Distributed Databases Principles and Systems*, McGraw-Hill, Inc., 1985.

[7] S. Ceri, B. Pernici and G. Wiederhold, "Optimization Problems and Solution Methods in the Design of Data Distribution", *Information Systems*, vol. 14, no. 3, pp 261-272, 1989.

[8] Sharma Chakravarthy, Jaykumar Muthuraj, Shamkant B. Navathe, "An Objective Function for Vertically Partitioning Relations in Distributed Databases and Its Analysis", *Distributed and Parallel Databases*, vol. 2, pp. 183-207, 1994.

[9] Sharma Chakravarthy, Jaykumar Muthuraj, Ravi Varadarajan, Shamkant B. Navathe, "An Objective Function for Vertical Partitioning Relations in Distributed Databases and its Analysis", *University of Florida Department of Computer and Information Sciences Technical Report*, 1992.

[10] Hong-Mei Chen Garcia and Olivia R. Sheng, "An Entity-Relationship-Based Methodology for Distributed Database Design: an Integrated Approach Towards Combined Logical and Distribution Designs", *Lecture Notes in Computer Science*, vol. 645, pp. 179-193, 1992.

[11] Wesley W. Chu, "Optimal File Allocation in a Multiple Computer System", *IEEE Transactions on Computer*, pp. 414-418, 1969.

[12] Wesley W. Chu and Ion Tim Ieong, "A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems", *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 804-812, Aug. 1993.

[13] Douglas W. Cornell and Philip S. Yu, "A Vertical Partitioning Algorithm for Relational Databases", *IEEE*, pp. 30-35, 1987.

[14] Douglas W. Cornell and Philip S. Yu, "An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases", *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 248-258, Feb. 1990.

[15] Ramez ElMasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, 2nd ed., The Benjamin/Cummings Publishing Company, Inc., 1994.

[16] Gary W. Hansen, James W. Hansen, *Database Management and Design*, 2nd ed., The Prentice-Hall International, Inc., 1996.

[17] Henry F. Korth, Abraham Salberschatz, *Database System Concepts*, 2nd ed. McGraw Hill International Edition, 1991.

[18] Xuemin Lin, Maria Orlawska and Yanchun Zhang, "A Graph Based Cluster Approach for Vertical Partitioning in Database Design", *Data and Knowledge Engineering*, vol. 11, pp. 151-169, 1993.

[19] Xuemin Lin and Yanchun Zhang, "A New Graphical Method for Vertical Partitioning in Database Design", *Australian Database Conference: Brisbane*, pp. 131-144, 1993.

[20] Fred R. McFadden, Jeffrey A. Hoffer, *Database Management*, 3rd ed., Benjamin/Cummings Pub. Co., 1991.

[21] C. Meghini and C. Thanos, "The Complexity of Operations on a Fragmented Relation", *ACM Transaction on Database Systems*, vol. 16, no. 1, pp. 56-87, Mar. 1991.

[22] J. Muthuraj, S. Chakravarthy, R. Varadarajan, S. B. Navathe, "A Formal Approach to the Vertical Partitioning Problem in Distributed Database Design", *University of Florida Technical Report*, 1992.

[23] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou, "Vertical Partitioning Algorithms for Database Design", *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 680-710, Dec. 1984.

[24] S. B. Navathe, K. Karlapalem and M. Ra, "A Mixed Fragmentation Methodology for Initial Distributed Database Design", *Journal of Computer and Software Engineering*, forthcoming vol. 3, no. 4, pp. 395-426, 1995.

[25] Shamkant B. Navathe and Minyoung Ra, "Vertical Partitioning for database Design: a Graphical Algorithm", *ACM SIGMOD*, pp. 440-450, June 1989.

[26] M. T. Ozsu, P. Valduriez, "Distributed and Parallel Database Systems", *In Handbook of Computer Science and Engineering*, A. Tucker (ed.), CRC Press, pp. 1093-1111 (Ch. 48), 1997.

[27] M. T. Oszu, P. Valduriez, "Distributed Database Systems: Where Are We Now?", *IEEE Computer*, vol. 24, no. 8, pp. 68-78, Aug. 1991.

[28] Tamer M. Ozsu, Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[29] Minyoung Ra, "Horizontal Partitioning for Distributed Database Design: a Graph-Based Approach", *Australian Database Conference Brisbane*, pp. 101-120, 1993.

[30] Andrew S. Tanenbaum, *Computer Networks*, 2nd ed., Prentice Hall International Editions, 1989.

[31] Yanchun Zhang, "On Horizontal Fragmentation of Distributed Database Design", *Australian Database Conference Brisbane*, pp. 121-130, 1993.

APPENDIX - SOURCE CODE

```
/*************************************************************************
HOBP.CPP file
This main file defines all the required implementations identified in
the HOBP.H classes.   It also includes the main function
*************************************************************************/
#include "hobp.h"

void main(void)
{
 class hobp h;
 for (int count=0; count<50; count++) { h.LoadData(count); h.Algo(); }
}

void hobp::Algo()
{
 cout << "The results for evaluating the sum cost" << endl;
 MinCost=EvaluateCost(Frag);
 ExpandNodes();
 cout << "1st fragment: ";
 for (int i=0;i<PREDICATES;i++)
   if (F1[i]!=ZERW)
     cout << "P" << i+1 <<" ";
 cout << endl;
 cout << "2nd fragment:";
 for (i=0;i<PREDICATES;i++)
   if (F2[i]!=ZERW)
     cout << "P" << i+1 <<" ";
 cout << endl << endl;
 SaveResultToFile();
}

void hobp::ExpandNodes(void)
{
 int i;
 struct node *tmp;
 for (i=0; i<=TRANSACTIONS; i++) {
    if (root==0) {
       root=GetNode();
       root->TransNum=i;
       E_node=root;
    }
    else {
       AssTrans[i-1]=1;
       AddPredicatesTfrag(i-1);
       cost=CalCost(i-1); //Step 4.
       tmp=GetNode();
       tmp->TransNum=i;
       NodeStack.push(E_node);
       if (cost < MinCost) { E_node->left=tmp; E_node=tmp; }
       else { E_node->right=tmp; PopTfrag(); }
    }// Else
 }// for
//Step 6:
 for (int j=0;j<PREDICATES;j++) TempF1[j]=Tfrag[j];
//TempF1=Tfrag
 for (j=0;j<PREDICATES;j++)
    if (TempF1[j]==ZERW) TempF2[j]=ONE;
    else TempF2[j]=ZERW; //TempF2=F-TempF1
 CurrentCost=EvaluateCost(TempF1)+EvaluateCost(TempF2);
 if (CurrentCost <= MinCost) {
    MinCost=CurrentCost;
    for (j=0; j<PREDICATES; j++) {
       F1[j]=TempF1[j];
       F2[j]=TempF2[j];
    }
 }
 else { // Step 7: of the algorithm
 if (!NodeStack.isStackEmpty()) { //pop the closest ancestor
      NodeStack.pop(E_node);
      E_node=E_node->right; //go to the right
      ExpandNodes(); //repeat the algorithm
    }
  }
}

void hobp::PopTfrag(void) //used to exclude the recently added
predicates
{
 if (!FragStack.isStackEmpty()) {
    FragStack.pop(tmpElement);
    for (int j=0;j<PREDICATES;j++)
       Tfrag[j]=tmpElement.frag[j];
 }
}

void hobp::PushTfrag(void)
{
 for (int j=0;j<PREDICATES;j++) tmpElement.frag[j]=Tfrag[j];
 FragStack.push(tmpElement);
}

void hobp::AddPredicatesTfrag(int tr)
{
 PushTfrag();
 for (int j=0;j<PREDICATES;j++)
   if (PUM[tr][j] != 0) Tfrag[j]=ONE;
}

CALVALUE hobp::CalCost(int tc)
{
 CALVALUE c=0.0;
 for (int j=0; j<PREDICATES; j++) {
    if (Tfrag[j]!=ZERW)
       c+=(PUM[tc][j] * freq[tc] * card[j]);
 }
 return(c);
}

struct node *hobp::GetNode()
{
 struct node *n;
 n=new struct node;
 assert(n!=0);
 n->TransNum=0;
 n->left=0;
 n->right=0;
 return n;
}

void hobp::LoadData(int c)
{
 int i, j;
 ifstream inData("Data.txt", ios::in);
 if (!inData) {
   cout << "Error in Opening file" << endl;
   exit(1);
 }
 cout << "Loading data for turn " << c <<", please wait..." << endl <<
endl;
 inData >> CARDINALITY >> TRANSACTIONS >> PREDICATES
>> OCCURRENCES;
 cout << "The cardinality of the relation is " << CARDINALITY <<
endl;
 cout << "The number of transactions is " << TRANSACTIONS <<
endl;
 cout << "The number of predicates is " << PREDICATES << endl;
 cout << "The maximum number of occurences is " <<
OCCURRENCES << endl;
 for (j=0;j<PREDICATES;j++) {
    Tfrag[j]=ZERW;
    Frag[j]=ZERW;
```

```
      F1[j]=ZERW;
      F2[j]=ZERW;
      TempF1[j]=ZERW;
      TempF2[j]=ZERW;
      AssTrans[j]=ZERW;
      card[j]=0;
  }
  for (i=0;i<TRANSACTIONS;i++) {
     freq[i]=0;
     AssTrans[i]=ZERW;
  }

  for(i=0;i<TRANSACTIONS;i++)
    for(int j=0;j<PREDICATES;j++)
      PUM[i][j]=0;

 // the PUM shows which transaction accessing which predicate
  cout << "Loading Predicate Usage Matrix ..." << endl;
  for (i=0;i<TRANSACTIONS;i++)
   for (j=0; j<PREDICATES;j++) {
     if (inData.eof()) break;
     inData >> PUM[i][j];
   }
  cout << "Loading Frequency of each transaction" << endl;
  for(i=0;i<TRANSACTIONS;i++)
    freq[i]=random(OCCURRENCES) + 10; // a transaction is issued
at least once
  cout << "Loading cardinality of each transaction when ";
  cout << "accessed to each predicate" << endl;
  for (j=0; j<PREDICATES;j++) {
    card[j]=(random(100) * 0.01 * CARDINALITY ) + 1;// number of
tuples returned when each predicates is applied (at least 1)
    Frag[j]=1; //the entire relation as a fragment
  }
  cout << endl << "Data Loaded" << endl << "Ready to run
simulation";
  cout << endl << endl;
  SaveDataToFile(c);
}

void hobp::SaveDataToFile(int c)
{
  int i, j;
  ofstream OutData("Result.TXT", ios::app);
  if (!OutData) {
    cout << "Could not open file" << endl;
    exit(1);
  }
  OutData << "Loading data for turn " << c << endl;
  OutData << "The cardinality of the relation is " << CARDINALITY
<< endl;
  OutData << "The number of transactions is " << TRANSACTIONS
<< endl;
  OutData << "The number of predicates is " << PREDICATES <<
endl;
  OutData << "The maximum number of occurences is " <<
OCCURRENCES << endl;
  OutData << "The Predicate Usage Matrix..." << endl;
  OutData << "\t";
  for (j=0;j<PREDICATES;j++)
     OutData << "P" << j+1 << "\t";
  OutData << endl;
  for (i=0;i<TRANSACTIONS;i++) {
   OutData << "T" << i+1 << "\t";
   for(j=0;j<PREDICATES;j++)
     OutData << PUM[i][j] << "\t";
   OutData << endl;
  }
  OutData << endl;
  OutData << "Frequency of each transaction:" << endl;
  for (i=0;i<TRANSACTIONS;i++) OutData << "T" << i+1 << "\t";
  OutData << endl;
  for(i=0;i<TRANSACTIONS;i++) OutData << freq[i] << "\t";
  OutData << endl << endl;
  OutData << "Cardinality of each predicate" << endl;
  for (j=0;j<PREDICATES;j++)  OutData << "P" << j+1 << "\t";
  OutData << endl;
  for(j=0;j<PREDICATES;j++)  OutData << card[j] << "\t";
  OutData << endl;
}

void hobp::SaveResultToFile(void)
{
  int i, j;
  ofstream OutData("Result.TXT", ios::app);
  if (!OutData) {
    cout << "Could not open file" << endl;
    exit(1);
  }
  OutData << endl ;
  OutData << "The results for evaluating the cost" << endl;
  OutData << "1st fragment: ";
  for (j=0;j<PREDICATES;j++)
    if (F1[j]!=ZERW)
      OutData << "P" << j+1 <<" ";
  OutData << endl;
  OutData << "2nd fragment: ";
  for (j=0;j<PREDICATES;j++)
    if (F2[j]!=ZERW)
      OutData << "P" << j+1 <<" ";
  OutData << endl << endl;
}

CALVALUE hobp::EvaluateCost(FRAGMENTS *frag)
{
  CALVALUE ho=0.0;
  for (int j=0; j<PREDICATES; j++)
    if (frag[j] != ZERW) ho+=(ReturnFreq(j) * card[j]);
  return (ho);
}

float hobp::ReturnFreq(int pr)
{
  float rf=0.0;
  for (int i=0;i<TRANSACTIONS;i++)
    if (AssTrans[i]==ONE) rf+=(PUM[i][pr] * freq[i]);
  return(rf);
}

void hobp::DelTree(struct node *t)
{
  if (t==0) return;
  DelTree(t->left);
  DelTree(t->right);
  delete t;
}

hobp::hobp()  //initialize the variables
{
  PUM=0;
  Frag=Tfrag=F1=F2=AssTrans=TempF1=TempF2=0;
  root=E_node=0;
  ofstream OutData("Result.TXT", ios::out);
  if (!OutData) {
    cout << "Could not open file" << endl;
    exit(1);
  }
  randomize();  // for the random generator
  clrscr();
  int i, j;
  ifstream inData("Data.txt", ios::in);
  if (!inData) { cout << "Error in Opening file" << endl; exit(1); }
  inData >> CARDINALITY >> TRANSACTIONS >> PREDICATES
>> OCCURRENCES;
  cout << "The cardinality of the relation is " << CARDINALITY <<
endl;
  cout << "The number of transactions is " << TRANSACTIONS <<
endl;
  cout << "The number of predicates is " << PREDICATES << endl;
  cout << "The maximum number of occurences is " <<
OCCURRENCES << endl;
```

```
 // Allocate single dimensional arrays
 Tfrag = new FRAGMENTS[PREDICATES];  assert(Tfrag != 0);
 Frag = new FRAGMENTS[PREDICATES];
assert(Frag != 0);
F1 = new FRAGMENTS[PREDICATES];
assert(F1 != 0);
 F2 = new FRAGMENTS[PREDICATES];     assert(F2 != 0);
 TempF1 = new FRAGMENTS[PREDICATES]; assert(TempF1 !=
0);
 TempF2 = new FRAGMENTS[PREDICATES]; assert(TempF2 !=
0);
 card =new int[PREDICATES];
 assert(card !=0);
 freq = new int[TRANSACTIONS];
 assert(freq != 0);
 AssTrans = new FRAGMENTS[TRANSACTIONS];
 assert(AssTrans != 0);
// Allocate two dimensional arrays
 *PUM =new int[TRANSACTIONS];       assert((*PUM) !=0);
 for (i=0;i<TRANSACTIONS;i++) { PUM[i]=new int[PREDICATES];
assert(PUM[i] !=0); }
}

hobp::~hobp()
{
 for (int i=0;i<TRANSACTIONS;i++) { delete [] PUM[i]; }
 delete [] PUM;    delete [] card;    delete [] Tfrag;   delete [] Frag;
delete [] F1;
 delete [] F2;     delete [] TempF1;    delete [] TempF2;    delete []
freq;
 delete [] AssTrans;
 DelTree(root);
}

/*************************************************************************
HOBP.H file
This header files defines all the required classes (variables and
interfaces) used for the simulation
*************************************************************************/
#include "stack.h"
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <values.h>
#include <time.h>  // for the randomize fct
#include <fstream.h>
#include <dos.h>

#define CALVALUE     float
#define FRAGMENTS    char
#define ZERW         '0'
#define ONE          '1'

//Number of transactions, predicates, cardinality of the relation and
the frequencies of each transaction in the simulation
int TRANSACTIONS, PREDICATES, CARDINALITY,
OCCURRENCES;
struct node {  //The element in the binary tree
  int TransNum;
  node *left, *right;
};

//a record is used instead to be used in the stack.
//Can not push an array to the stack but a structure it is feasable
struct element {
  FRAGMENTS frag[30];
};

class hobp {
  public:
    hobp();  //initialize the variables
    ~hobp();
    void Algo(void);
    void LoadData(int); //load the data
  private:
    void AddPredicatesTfrag(int); //copy predicates to transaction i
    void ExpandNodes(void);
    CALVALUE CalCost(int); //calculate the cost of including the
predicates of transaction tc
    void SaveDataToFile(int);
    void SaveResultToFile(void);
    void DelTree(struct node *);
    CALVALUE EvaluateCost(FRAGMENTS *);
    float ReturnFreq(int);
    int **twoDimInt(int, int);
    float **twoDimFloat(int, int);
    struct node *GetNode();
    CALVALUE cost, CurrentCost, MinCost; //used for calculating
the cost
    struct node *root, *E_node, *LeftNode, *RightNode; //for the
construction of the tree

//the Tfrag, final fragments F1, F2 and temporary fragments used in
the
//simulation and frag represents the entire relation.  AssTrans is
assigned transactions
    FRAGMENTS *Tfrag, *F1, *F2, *Frag,
              *TempF1, *TempF2,  //size of PREDICATES
              *AssTrans; //size of transactions

//Structures used for the simulating an example
    int **PUM; // predicate usage matrix
    int *freq; //frequencies of each transaction in an array
TRANSACTIONS
    int *card; //cardinality of each predicate

//A stack is used in the simulation
    void PushTfrag(void);//sets the element structure to be pushed
in stack
    void PopTfrag(void);//pop the element from stack and sets the
                        //respective values
    struct element tmpElement;
    Stack<struct element> FragStack; //for restoring previous state
of Tfrag
    Stack<struct node *> NodeStack; //for the backtracking in the
tree
};

/*************************************************************************
STACK.H file
Stack.h data type uses the class list in List.h to be implemented
*************************************************************************/
#ifndef STACK_H
#define STACK_H
#include "list.h"

template <class STACKTYPE>
class Stack : private List<STACKTYPE> {
  public:
    void push(const STACKTYPE &d) { insertAtFront(d); }
    int pop(STACKTYPE &d) { return removeFromFront(d); }
    int isStackEmpty() const { return isEmpty(); }
};

#endif

/*************************************************************************
LIST.H file
The basic interfaces and implentations routines used for the stack
are defined in this file
*************************************************************************/
#ifndef LIST_H
#define LIST_H
#include <iostream.h>
#include <assert.h>

template <class NODETYPE>
class ListNode {
   friend class List<NODETYPE>; //make List a friend
  public:
    ListNode(const NODETYPE &); //constructor
    NODETYPE getData() const;  //return the data in the node
```

```
  private:
    NODETYPE data;
    ListNode *nextPtr;
};

template <class NODETYPE>
class List {
  public:
    List();
    ~List();
    void insertAtFront(const NODETYPE &);
    void insertAtBack(const NODETYPE &);
    int removeFromFront(NODETYPE &);
    int removeFromBack(NODETYPE &);
    int isEmpty() const;
  private:
    ListNode<NODETYPE> *firstPtr, *lastPtr;//pointer to first and
last node
    ListNode<NODETYPE> *getNewNode(const NODETYPE &);
//utility
};

template <class NODETYPE> //constructor
ListNode<NODETYPE>::ListNode(const NODETYPE &info)
{
 data=info;
 nextPtr=0;
}

template <class NODETYPE>
NODETYPE ListNode<NODETYPE>::getData() const { return data;
}

template<class NODETYPE> //Constructor
List<NODETYPE>::List() {firstPtr=lastPtr=0; }

template<class NODETYPE> //Destructor
List<NODETYPE>::~List()
{
 if (!isEmpty()) {  //List is not empty
  ListNode<NODETYPE> *currentPtr = firstPtr, *tempPtr;
  while (currentPtr !=0) { //delete remaining nodes
    tempPtr=currentPtr;
    currentPtr=currentPtr->nextPtr;
    delete tempPtr;
  }
 }
}

template<class NODETYPE> //Insert a node at the front of the list
void List<NODETYPE>::insertAtFront(const NODETYPE &value)
{
 ListNode<NODETYPE> *newPtr = getNewNode(value);
 if (isEmpty()) // list is empty
   firstPtr=lastPtr=newPtr;
 else {
   newPtr->nextPtr=firstPtr;
   firstPtr=newPtr;
 }
}

template<class NODETYPE> //Insert a node at the back of the list
void List<NODETYPE>::insertAtBack(const NODETYPE &value)
{
 ListNode<NODETYPE> *newPtr=getNewNode(value);
 if (isEmpty()) // list is empty
   firstPtr=lastPtr=newPtr;
 else {
   lastPtr->nextPtr=newPtr;
   lastPtr=newPtr;
 }
}

template<class NODETYPE> //delete a node from the front of the
list
int List<NODETYPE>::removeFromFront(NODETYPE &value)
{
 if (isEmpty()) return 0;
 else {
  ListNode<NODETYPE> *tempPtr=firstPtr;
  if (firstPtr==lastPtr)
    firstPtr=lastPtr=0;
  else
    firstPtr=firstPtr->nextPtr;
  value=tempPtr->data;
  delete tempPtr;
  return 1;
 }
}

template<class NODETYPE> //delete a node from the back of the
list
int List<NODETYPE>::removeFromBack(NODETYPE &value)
{
 if (isEmpty()) return 0;  //delete successful
 else {
  ListNode<NODETYPE> *tempPtr=lastPtr;
  if (firstPtr==lastPtr)
    firstPtr=lastPtr=0;
  else {
    ListNode<NODETYPE> *currentPtr=firstPtr;
    while (currentPtr->nextPtr != lastPtr)
          currentPtr=currentPtr->nextPtr;
    lastPtr=currentPtr;
    currentPtr->nextPtr=0;
  }
  value=tempPtr->data;
  delete tempPtr;
  return 1;  //delete successful
 }
}

template<class NODETYPE> //is the list empty?
int List<NODETYPE>::isEmpty() const { return firstPtr == 0; }

template<class NODETYPE> //Return a pointer to a newly
allocated pointer
ListNode<NODETYPE> *List<NODETYPE>::getNewNode(const
NODETYPE &value)
{
 ListNode<NODETYPE> *ptr= new ListNode<NODETYPE>(value);
 assert(ptr != 0);
 return ptr;
}

#endif
```