REDESIGN OF DISTRIBUTED RELATIONAL DATABASES

A THESIS
Presented to
The Faculty of the Division of Graduate Studies

by

Kamalakar Karlapalem

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
November 1996

Copyright © 1996 by Kamalakar Karlapalem

REDESIGN OF DISTRIBUTED RELATIONAL DATABASES

Approved:

____

Prof. Shamkant B. Navathe

____

Prof. Mostafa Ammar

____

Prof. Edward Omiecinski

____

Prof. Leo Mark

____

Prof. Sridhar Narasimhan
(School of Management)

Date Approved by Chairman ____________

To My Grandfather
Sri Karlapalem Lakshmi Narasimham

అభ్యసానుసరీ విద్యా బుద్ధిహి కర్మాను సారిణీ

ఉద్యోగానుసరీ లక్ష్మీహ్ ఫలం భాగ్యానుసారీచ

Sanskrit

విద్య నేర్చుకొనుట అభ్యాసము చేయుటపై ఆధారపడియుండును.
బుద్ధి (జ్ఞానము) కర్మా (గతములో నడవడి బట్టీ) పై ఆధారపడియున్నది.
ధన ధాన్య సంపదలు కష్ట పడుట వలన కల్గును.
ఫలము (విజయము) అద్రుష్టము మీద ఆధారపడియున్నది.

Telugu

Acquiring knowledge (learning) is dependent upon constant drilling (practice).

Wisdom (intellect) is a result of the past acts (karma).

Endeavor (effort; hard work) leads to money, material, and possessions.

Good luck (destiny; fortune) leads to success.

ACKNOWLEDGEMENTS

Prof. Navathe's support, guidance, and encouragement has been instrumental for this work. I am very much grateful for all the help that he has provided to me over last five years, and I do owe him a lot. I consider myself to be very fortunate to have him as my advisor. I thank Prof. Ammar for his suggestions at crucial times of this work which helped me to clarify things, and put me in the right direction, Prof. Omiecinski for his comments and suggestions, and Profs. Narasimhan and Mark for being in the committee and participating in the discussions. Jorg and George helped me with the implementation of Markovian decision analysis algorithms. This work would not have been possible with out the financial support from University of Florida, Georgia Tech, National Science Foundation grant number IRI 8716798, Hewlett Packard, and Digital Equipment Corporation.

After being in three places over four years, I have come to know lot of people who have made my life easy and happy. I thank the D^3T team, Dr. Ravi Vardarajan, Pedro, and MinYoung for the stimulating discussions, Sharon Grant was always helpful at UF, the ALLBASE/STAR team at Hewlett Packard, David, Brad, John, Leigh-Ann, and Susan for all their support during my stay at Cupertino, and the OIT team at Georgia Tech, Ruth Strausser, Art Vanderberg, and Scott for giving me the material for the case study.

Special thanks to Magdi, Asterio, and Kiran, my colleagues at Georgia Tech, for all their help, and making my stay here comfortable and enjoyable. There were innumerable persons at all these three places who have helped me one way or other, and to all of them my sincere thanks.

This thesis would not have been possible without the motivation and support from my parents, sisters, and my wife Santhi. My utmost thanks to them.

CONTENTS

ACKNOWLEDGEMENTS v

SUMMARY xiii

1 INTRODUCTION 1

1.1 Background 1

1.2 Basic Concepts 2

1.3 The Distributed Database Design Process 4

1.4 Redesign of distributed databases 5

1.4.1 When to redesign? 5

1.4.2 How to come up with the new design? 6

1.4.3 How to materialize the new design? 7

1.5 Application Processing Center 7

1.6 Contributions 8

1.6.1 Scope and Limitation 10

1.7 Application of the Proposed Approach 10

2 RELATED WORK 12

2.1 General Comments 12

2.2 Distributed database design 12

2.2.1 Fragmentation 12

2.2.2 Allocation 14

2.3 Redesign of distributed databases 15

2.4 Response time analysis 16

2.5 Markovian Decision Analysis 16

2.6 Summary 17

3 OVERVIEW OF THE MIXED FRAGMENTATION METHODOLOGY 18

3.1 General Comments 18

3.2 Alternative approaches for initial distributed database design 18

3.2.1 Mixed Fragmentation Methodology 21
3.3 Grid creation 23
3.3.1 Vertical partitioning for grid creation 23
3.3.2 Horizontal partitioning for grid creation 26
3.3.3 Procedure Horizontal-Partitioning 29
3.3.4 Grid Cells 34
3.4 Summary 35

4 REPRESENTATION SCHEME FOR MIXED FRAGMENTS 36
4.1 General Comments 36
4.2 Representation Scheme for Mixed Fragments 36
4.2.1 Characteristics of and valid operations on grid cells 38
4.2.2 Mapping of grid cells to transactions 41
4.3 Grid Optimization 42
4.3.1 Cost model for merging 43
4.3.2 Merging algorithm 44
4.3.3 The case of overlapping fragments 46
4.3.4 The case of contained in fragments 49
4.4 Summary 51

5 MATERIALIZATION OF DISTRIBUTED DATABASES 52
5.1 General Comments 52
5.2 Two approaches to materialize a new design 52
5.3 Representation of Distribution Designs 53
5.3.1 Intersection of fragmentation schemes 57
5.4 Approaches for Materialization 61
5.4.1 Materialization of a redesigned distributed relational database by automatic generation of SQL commands 62
5.4.2 Materialization with operations on fragments 65
5.5 Extension 73
5.6 Cost Model 75
5.7 Summary 79

6 **EFFICIENT MATERIALIZATION BY USING THE QUERY GENERATOR APPROACH** **80**
6.1 General Comments 80
6.2 Multiple Query Optimization 80
6.3 Example Illustrating use of MQO 81
6.4 MQO Algorithm to Materialize Distributed Database Design 95
6.5 Performance Improvement 100
6.6 Summary 101

7 **AVERAGE TRANSACTION RESPONSE TIME ESTIMATION** **102**
7.1 General Comments 102
7.2 The Distributed Database Environment 103
7.3 Transaction Characteristics 103
7.4 Estimating the Probability of Contention 105
7.5 A Model For The Locking Mechanism 107
7.6 Simulation Model 108
7.7 Experiments 111
7.7.1 Effect of Data Access Time 111
7.7.2 Effect of Arrival Rate of Transactions 113
7.7.3 Evaluation of the Optimal Percentage of Distributed Transactions . 114
7.7.3.1 Practical Applicability 119
7.7.4 Effect of Number of Data Accesses 121
7.8 Summary 123

8 **A DISTRIBUTED DATABASE REDESIGN METHODOLOGY** **125**
8.1 General Comments 125
8.2 A Model for the Application Processing Center Scenario 125
8.3 Application Classes 129
8.3.1 Change points and Execution Duration 130
8.3.2 Calculating the Probability of Change 131
8.4 Generation of Candidate Distributed Database Designs 132
8.5 Calculating the Cost Values 134
8.6 Optimal Policy Vector Generation 135
8.6.1 Modeling of Redesign Problem 135

8.7 Summary 141

9 MARKETING AND RECRUITING SYSTEM: A CASE STUDY 142

9.1 General Comments 142

9.2 Distributed Database Environment 142

9.3 Database Relations and Application 143

9.3.1 Defining Classes of Applications 146

9.3.2 Transaction and Relation Characteristics 147

9.4 Distributed Database Designs 148

9.4.1 Prospects 149

9.4.2 Recruiters 155

9.4.3 Other Relations 158

9.4.4 Fragmentation and Allocation Schemes for On-line and Batch Applications 158

9.5 Application Processing Center Scenario 160

9.6 Cost Values Calculation 161

9.7 Summary 166

10 CONCLUSIONS 167

A BREAK, FORM AND CLUSTER OPERATIONS 173

B OPTIMAL POLICY VECTOR GENERATION DETAILS 175

C SELECT STATEMENT DESCRIBING DATA ACCESSED BY TRANSACTIONS 180

BIBLIOGRAPHY 193

VITA 194

LIST OF TABLES

1 Representation of Current Design (F, A). 56
2 Representation of New Design (F', A'). 60
3 Minimal Cover for Fragments in the New Design 60
4 Intersect for Fragments in the Current Design 61
5 Workload Parameters 109
6 System Parameters 110
7 Maximum Data Transfer for Distributed Transaction Processing 120
8 A Redesign Policy 127
9 Redesign Policies for 3 Applications and 3 Designs 138
10 Relation Recruiters 143
11 Relation Prospects 144
12 Relation Events 145
13 Relation ProsRecr 145
14 Schedule for Applications and Transactions Executed During the Day . . . 146
15 Cardinalities of the Relations 147
16 Selectivity of Predicates in the Relations 148
17 Transaction Initiation Frequencies 148
18 Grid Cells of Relation Prospects with Online-Transactions (A1). 150
19 Distributed Database Design for relation Prospects (A1) 152
20 Grid Cells of Relation Prospects with Online-Transactions (A2). 153
21 Distributed Database Design for relation Prospects (A2) 153
22 Grid Cells of Relation Prospects with Batch Transactions 154
23 Distributed Database Design for relation Prospects (A3) 154
24 Grid Cells of Relation Recruiters for Class of Online-Transactions (A1). . . 156
25 Distributed Database Design for relation Recruiters (A1) 156
26 Grid Cells of Relation Recruiters for Class Online-Transactions (A1). 157
27 Distributed Database Design for relation Recruiters (A2) 157
28 Grid Cells of Relation Recruiters with Batch-Transactions 158
29 Distributed Database Design for relation Recruiters (A3) 158

LIST OF FIGURES

1 Alternatives for fragmentation and allocation 19
2 Distributed database design tool reference architecture 20
3 Steps in Mixed Fragmentation Methodology 22
4 Transaction Specification 23
5 Attribute Usage Matrix 24
6 Attribute Affinity Matrix 25
7 Vertical fragments generated by graph-theoretic algorithm 26
8 Predicate Usage Matrix 28
9 Predicate Affinity Matrix 29
10 Clustering of predicates using graph-theoretic algorithm 31
11 Non-overlapping horizontal fragments generation 33
12 Representation of grid cells 37
13 Mapping of Transactions to the Grid Cells 42
14 Overlapping and Contained-in Fragments 47
15 Example of overlapping and contained-in mixed fragments 48
16 Relation Employee 54
17 Grid Cells of the Relation Employee 55
18 Current Design (F, A) 56
19 Current Fragmentation Scheme (F, A) 57
20 New Fragmentation Scheme (F', A') 58
21 Representation of Fragmentation Scheme (F', A') 59
22 A Hierarchy of Fragment Operations 68
23 Initial Query Graphs 85
24 Step 1 of Multiple Query Optimization 86
25 Step 2 of Multiple Query Optimization 89
26 Combined optimized query graph 92
27 Local and Distributed Transaction Phases 104
28 Simulation Model For Lock Contention 108
29 Effect of Data Access Time on Average Transaction Response Time 112
30 Effect of Arrival Rate on Average Transaction Response Time: I 113

31 Effect of Arrival Rate on Average Transaction Response Time: II 115
32 Evaluation of Optimal Percentage Points with Arrival Rate 10 trns/sec . . . 116
33 Evaluation of Optimal Percentage Points with Arrival Rate 15 trns/sec . . . 117
34 Optimal Distr Trans Percentage Over CommDelay 118
35 Effect of Change in Number of Local Data Accesses 122
36 Effect of Change in Number of Distributed Data Accesses 123
37 Discrete Markov Process and Redesign Policy 128
38 Discrete Markov Process Defined by the Optimal Policy Vector 140
39 Candidate Distributed Database Design (D1) for On-Line Applications (A1). 159
40 Candidate Distributed Database Design (D2) for On-Line Applications (A2). 159
41 Candidate Distributed Database Design (D3) for Batch Applications (A3). . 160
42 Long Run Discrete Markov Process For Application Processing Scenario I. 164
43 Long Run Discrete Markov Process For Application Processing Scenario II. 165
44 A Distributed Database Design/Redesign Tool Architecture. 170

SUMMARY

Distributed relational databases are becoming a reality now; hence in order provide an efficient support for the applications, designing distributed databases becomes an issue of paramount importance. Distributed database design consists of a fragmentation and an allocation schema. The fragmentation schema maps the relations of the distributed database to the fragments of data at individual sites. The allocation schema maps the fragments to the different sites of the distributed database environment. Changes in the distributed database environment over a period of time necessitate the redesign of distributed databases.

This thesis develops a practical solution for the problem of design and redesign of distributed databases in the application processing center environment. The mixed fragmentation methodology is used for generating the fragmentation scheme for the distributed databases. A representation scheme for the grid cells and the notion of a regular fragment is developed. This representation scheme is used for developing the algorithms for merging the grid cells and materialization of the distributed databases. Two approaches (Query Generator Approach and Operator Method) have been developed for the materialization of distributed databases. A cost model is developed to compare these two approaches. Multiple query optimization techniques are used to increase the efficiency of the query generator approach. The cost of applications may be approximated in terms of the average cost of transactions that constitute an application. A simulation model is presented to estimate the average transaction response time in the distributed database environment. A set of experiments were run to evaluate the effect of changes in some parameters (like arrival rate, communication delay, number of data accesses) on the average transaction response time.

The Markovian decision analysis technique is used to develop a methodology for redesign of distributed databases in the application processing center environment. This technique generates an optimal policy that associates a specific distributed database design with each execution of the application. This technique guarantees that this policy will process all the applications efficiently in the long run. A case study of a real life application processing scenario is presented to illustrate this methodology.

CHAPTER 1

INTRODUCTION

1.1 Background

Relational database system technology has been under development for more than two decades. At present there are many efficient and high throughput relational database systems. During these two decades most of the high throughput database processing was done on either hierarchical database management system (e.g., IMS) or network database management system (e.g., IDMS and DMS1100), or flat file systems. The relational database technology is at a stage where it can efficiently process high throughput database applications. There are various reasons for this:

- relational database system technology has matured, there are very efficient commercial implementations of relational database systems, for example ORACLE (of ORACLE Corp.), INGRES (of Ask Systems), DB2 (of IBM), Rdb (of Digital), ALLBASE (of HP), etc. These systems are being widely used for day to day data processing.

- powerful multiprocessor based high end workstations have further enhanced the power of the relational database technology as a clear separation between the logical and physical levels facilitates efficient parallel processing of database operations. E.g., ORACLE on KSR1 machine.

- relational databases support non-navigational data access (meaning that the relationships between the database instances are not hard coded in the implementation model) which enables easy conceptualization and implementation of the distributed relational database system.

- with the advent of standards for remote data access (RDA) and SQL access, it is now possible to access data from different relational database systems. E.g., an application can access tuples of relations from ORACLE and INGRES databases.

- there will be a movement from centralized mainframe processing to distributed relational database processing on local area networks with high end workstations. This will happen without losing the throughput for the transaction oriented database applications. In fact, this environment will be scalable and more adaptable to the changes in the application processing requirements.

- Client-Server architecture helps in keeping main database functionality on the server and applications can be distributed across clients. We are likely to see a number of variations (e.g. one client-many servers, etc.) that will be limited cases of distributed database processing.

Most of the work in distributed relational database systems has concentrated on the systems aspect; not much has been done on the design and management of distributed databases. It is imperative that this problem will confront the users of the distributed database technology fairly soon.

1.2 Basic Concepts

A distributed database is a collection of data that belongs logically to the same system but is physically spread over the sites of a computer network. The system that manages the distributed databases is known as a distributed database management system. An implementation of a distributed database management system will consist of a set of cooperating database management systems each of which manages data physically located at a site. The layer of software that supports this cooperation forms the distributed database management system. A distributed database system refers to the distributed database management system together with the distributed databases. A homogeneous distributed database system is one wherein all the database systems at different sites are the same. A *Distributed Relational Database System* is a distributed database system wherein all of the database systems are relational and homogeneous in that they are essentially (versions of) the same DBMS from the same vendor. A distributed database environment refers to the distributed database system, network, and the computers at various nodes.

Distributed database systems are becoming a reality now, a number of computer manufacturers and database vendors are releasing distributed relational database system products. There is a need to support high performance on-line and batch processing of high volume transactions. In any distributed database there are two main factors that affect the

performance of the applications, namely, *number of disk accesses* and *data transfer cost or communication delay*. For a given distributed database environment the performance of the applications is dependent on the design of the distributed database. There are two ways by which performance of the applications can be improved by the distributed database design.

Fragmentation. Fragmentation is the process of clustering relevant columns and tuples accessed by an application into a fragment. This reduces the amount of irrelevant data accessed by the application thus reducing the number of disk accesses. The result of the fragmentation process is a set of fragments defined by a fragmentation scheme. In relational databases, a fragment can be either vertical, horizontal or mixed. A vertical fragment of a relation is a projection of a set of columns (including a key) of the relation. A horizontal fragment is a selection of a set of tuples of the relation. A mixed fragment is either a selection of tuples from a projection of a set of columns of a relation or a projection of a set of columns from a set of selected tuples of the relation.

Allocation. Allocation is the process of placing the fragments generated by the fragmentation scheme at the sites of the distributed database systems so as to minimize the data transfer cost and the number of messages to process a given set of applications. The result of an allocation process is an allocation scheme.

The fragmentation and allocation schemes define the distribution design for a distributed database. The global conceptual schema is produced by using the standard techniques of requirements analysis, view analysis and integration [BCN91]. In addition to the standard requirements analysis for the conceptual design of a distributed database, data is collected about the transaction characteristics for the distribution design like: the columns they access, the frequency with which they are initiated from a site, the predicates defining the tuples that the transactions access, and the applications the transactions belong to. Note that different applications initiate different sets of transactions and the distribution design is dependent on the columns and tuples of the relations accessed by the transactions.

1.3 The Distributed Database Design Process

The distributed database design process consists of four phases:

Initial Design. The initial design deals with generating the fragmentation and allocation scheme for the distributed database by making use of the transaction characteristics collected at the requirements analysis phase so as to optimize the global transaction processing costs.

Final Design. The final design corresponds to the actual production database that is used to support the applications. The initial design is implemented on a distributed database system and is evaluated for its performance. This implemented database is fine tuned based on the distributed connectivity, client-server architecture scenarios and physical characteristics such as the amount of main memory and disk space available. Once this implemented database is stable, it becomes the final design that is used to support the applications.

Redesign. In a distributed database environment over a period of time, there may be a degradation in the performance of the applications accessing the distributed database. This is due to the changes in the distributed database environment classified as follows:

Physical. The changes pertaining to the physical systems of the distributed database environment are classified as the physical changes. Types of physical changes include: change in network topology, new nodes being added, changes in "processing speed or power" of a node, or placing a more efficient disk pack at a node.

Logical. Changes in the distributed database environment that are not physical in nature are termed as the logical changes. Logical changes include: migration of applications/transactions from one node to another, change in the logical model of the database, change in the frequency of execution of an application, adding or deleting applications being executed on the distributed database.

These changes necessitate redesign of the distributed databases in order to keep the performance of the applications/transactions being supported by the distributed database system from degrading. In this phase, the changes in the distributed database

environment are taken into consideration to generate new fragmentation and allocation schemes. The redesign problem has been classified in [WN86] into **limited redesign** and **total redesign**. Limited redesign is the case where the allocation scheme changes but not the fragmentation scheme. Total design on the other hand considers the case where both the fragmentation and the allocation schemes change. The focus of this research is on the total redesign of the distributed databases as elaborated in the next section.

Materialization of Redesign. Materialization of a new design based on the current design for an existing populated distributed relational database after a total redesign has been an open research problem. *Materialization of a distributed database design is defined as a process of global restructuring of the populated local databases in order to achieve conformance with the logical definition of the distributed database.* Two elegant solutions to this problem are presented in this thesis.

Of the above four phases of the distributed database design process, much work has been done on the *initial design* problem, a limited case of the redesign problem has been addressed, and some aspects of the materialization problem have been addressed with respect to the limited redesign problem.

1.4 Redesign of distributed databases

The problem we address in this thesis is that of *Redesign of Distributed Relational Databases*. This problem can be broadly divided into three parts, namely:

1.4.1 When to redesign?

There are three scenarios which require the redesign of the distributed database. The first scenario is that of **corrective redesign**, the case where there is a deterioration in the application's performance because of the changes in the distributed database environment that had already taken place. Corrective redesign requires evaluating these changes in the distributed database environment to generate a new distribution design. The second scenario is that of **preventive redesign**, the case where the distributed database administrators know beforehand the changes that will take place in the distributed database environment. This allows them to switch to a new design before the actual change occurs, thus preventing

a deterioration in the performance of the applications. The third scenario is the case when the distributed database administrator wants to do a **adaptive redesign**. That is, monitor some parameters in the distributed database environment and initiate a redesign whenever the value of the parameter falls below certain level. An example of such a parameter is the transaction response time.

The corrective redesign is the easiest to do. This is because the distributed database administrator knows beforehand about the changes that took place in the distributed database environment, and uses this information to redesign. The preventive redesign is tougher than corrective redesign when the details about the types of changes taking place are known, *but not when these changes take place*. Adaptive redesign is the toughest to do. This is because it is difficult to detect relevant changes in the parameters being monitored. A distributed relational database environment is complicated; there are many interdependent logical and physical parameters and it is very difficult even to analyze the environment so as to detect a change in some parameter that reflects deterioration in the performance of the applications on the distributed database system. In centralized database systems, periodic tuning (like changing the buffer sizes, redefining indexes) is undertaken to enhance the performance of the database system to support the applications more efficiently. In the distributed database system, not only does the DBA have to undertake the periodic tuning by each of its local database systems, but also the global reorganization of data so as to efficiently support the applications. This is pertinent to the case where there are physical and logical changes taking place in the distributed database environment.

1.4.2 How to come up with the new design?

The problem here is to take into consideration the changes in the distributed database environment, and to modify the current design so as to efficiently support distributed database applications. The initial design methodology which has been developed is so efficient that it can be used as a methodology for redesigning distributed databases. No single algorithm can take into consideration all the different types of changes in the distributed database environment and incrementally modify the current design. Even if this was possible, the complexity of this algorithm would be much worse than that of the initial design methodology.

1.4.3 How to materialize the new design?

The initial design methodology is used to generate a new design based on the changes in the distributed database environment. This design needs to be materialized. Materialization means global restructuring of component databases of the distributed database so as to conform to the new fragmentation and allocation schemes as defined by the new distribution design. This is done by executing a set of algorithms. In this thesis a set of algorithms are developed and their correctness is established.

1.5 Application Processing Center

An application processing center is a data center which has computer resources, database systems that supports processing of a set of different classes of applications. Traditionally, these application processing centers used a centralized database management system like IMS on a mainframe. However, the cost of maintaining mainframes has increased, and the reliability, processing power of local area networked high end workstations has increased. With that there will be a move towards shifting application processing from centralized database systems to distributed database systems.

This would require a design of a distributed database that is optimal for all the classes of applications. But the optimality of a single design for all classes of applications cannot be guaranteed. Due to the different data and processing requirements for each class of applications, there will exist an optimal distributed database design for each class of applications which will outperform a single optimal distributed database design for all classes of applications. Moreover, not all applications from different classes are executed at the same time. Typically, there is a schedule for executing the applications, where each class of applications is processed during some intervals of time with some probability. Some applications which are always executed (i.e. with probability = 1), are known as *perpetual applications*. Applications which are executed at some specific intervals of time are known as *non-perpetual applications*.

A typical example of an application processing center is that of a bank or insurance company that processes various classes of high transaction oriented database applications. Examples of perpetual applications are teller machine transactions, or insurance claims. Examples of non-perpetual applications are those executed daily like day-time applications for processing loans, or creating new accounts, versus night time batch applications for

generating consolidated teller machine activities. Other applications include weekly salary processing, monthly account statement generation, and yearly W-2 form generation.

An application processing center is the life line for the business organization. It is basically a production center that processes information for the organization. Hence applications being executed in this center are well defined, well designed and well tested. It is very critical for the organization that these applications be executed in a timely and efficient manner. It is for this reason that this scenario is used to develop our redesign methodology. The general distributed database processing scenario is very complicated because of the dynamic changes which make it very difficult to decide when to redesign. Also, there are ill defined applications that are being processed in an ad-hoc fashion which complicate the problem of when to redesign.

1.6 Contributions

Consolidation of Mixed Fragmentation Methodology

The mixed fragmentation methodology was developed as part of the Distributed Database Design Tool Project (D^3T) at the University of Florida. Early work on mixed fragmentation methodology consisted of development of graphical algorithms for vertical and horizontal fragmentation. Grid cells were generated by simultaneously applying both vertical and horizontal fragmentation schemes on the relation. Some heuristic based algorithms were developed for vertical and horizontal merging of grid cells based on the cost of accessing the grid cells. In this thesis, a representation scheme for grid cells, and a notion of well formed expression and regular fragment was developed. This concept of regular fragment is used in developing a single algorithm for merging grid cells to form regular mixed fragments and prove its correctness. The representation scheme is also used to develop a solution to the problem of materialization of the distributed databases.

Materialization of Distributed Databases

Any redesign of a distributed database results in a change of fragmentation scheme and/or allocation scheme. This requires that the new fragmentation and/or allocation scheme be materialized. The mixed fragmentation methodology provides a basis for an intuitive solution to the materialization of distributed databases problem. The representation scheme makes the task of proving the correctness of the algorithms simple. Two approaches for materializing the distributed databases were developed. The first approach was to generate

a set of SQL "SELECT ..." statements to define the fragments of the distributed database and execute them on the distributed database system. A query generator algorithm is developed and its correctness is proved. The task of materialization rests solely on the distributed database management system. In the second approach (operator method) a set of operations are defined on the fragments (like split, merge, move, etc.). The materialization is done by generating a sequence of operations on the fragments to be executed at each site of the distributed database environment. In this thesis, algorithms were developed to generate these sequence of operations, and their correctness was proved. A cost model is developed to calculate the time required to materialize the distributed database for both the approaches. The operator method is shown to perform better than query generator approach when no proper indexes are defined on the fragments. The efficiency of the query generator approach was increased to make it comparable to the operator method by developing a multiple query optimization algorithm.

Estimation of Average Transaction Response Time

An event-based simulation model is developed to estimate the average transaction response time in a distributed database environment. This model takes into account the cyclic dependence between the lock contention and the average transaction response. A set of experiments were conducted to analyze the effect of changes in different parameters of the distributed database environment on average transaction response time. The tradeoff between the amount of time spent of data access and communication delay is analyzed. This simulation model is used to estimate the cost of executing a set of applications on a given distributed database design.

A Distributed Database Redesign Methodology

A redesign methodology is developed for the application processing center scenario. An optimization technique (Markovian Decision Analysis) is used to generate an optimal redesign policy that specifies the distributed database design that needs to be used for each application initiated. This optimization technique guarantees that this policy will process all the applications efficiently, and will require a redesign only if it is beneficial on the long run. This solution for the redesign problem is based on the mixed fragmentation methodology, the algorithms to merge grid cells, the simulation model to estimate the average transaction response time, and the algorithms to materialize the distributed databases. A case study is presented to illustrate the applicability of this methodology to a set of applications of a system similar to that in use at Georgia Tech.

1.6.1 Scope and Limitation

The techniques presented in this thesis are applicable only if mixed fragmentation methodology is used for distributed database design and redesign. The representation scheme of mixed fragments is based on the mixed fragmentation methodology. The allocation of mixed fragments is still an open problem. Therefore, the distributed design methodology based on mixed fragmentation approach is an incomplete methodology. A major limitation is that there is no true distributed relational database management system in the commercial world that can be used to experimentally test the effectiveness of techniques presented in this thesis. This is the reason that a simulation model had to be developed for estimating the average transaction response time. In this model, a pessimistic concurrency protocol is used (i.e. the contending transactions are blocked). The estimation of average transaction response time with optimistic or semi-optimistic concurrency control protocols is not addressed. The Markovian decision analysis is based on the assumption of the Markov process and the knowledge of discrete change points. But application processing scenario in reality may be a continuous or semi-Markovian processes. This thesis does not address these scenarios. Though a cost model is presented to calculate the cost of materialization of a redesign, experimental studies are still needed to observe the actual time to materialize the distributed database design using both the methods. The system issues for implementing the operator method to materialize the distributed database are not addressed.

1.7 Application of the Proposed Approach

The early work on distributed database design was done during the 1980-1983 time period. During that time a set of algorithms for vertical fragmentation, horizontal fragmentation and allocation were developed. But over the last decade little work was done in this area. The problems of horizontal fragmentation, allocation, and redesign are the toughest problems in this area. There was work done in limited redesign and complexity analysis of algorithms for materializing limited redesign. The mixed fragmentation methodology along with the problems addressed in this thesis forms the basis for providing a viable solution for distributed database design and redesign methodology in application processing center environment.

The solution consists of the following steps:

- The algorithms developed for mixed fragmentation are used to generate a set of candidate designs for a set of applications.
- The simulation model is used to estimate the average transaction response time for each class of applications executed on each of the candidate distributed database designs.
- The algorithms for materialization are used to calculate the cost of redesign from one candidate distributed database design to another.
- The Markovian decision analysis is used to generate the optimal policy vector that specifies the design which is to be used by application that is initiated.

The operator method generates a set of relation transfer operations that can used to evaluate the efficiency of algorithms that schedule the redistribution of data. The algorithms to materialize the distributed databases can be used independent of the redesign methodology presented in this thesis. As this is the first piece of work in total redesign of distributed databases, the solutions provided in this thesis are subject to further analysis and optimization. A real life application processing scenario is used to demonstrate the solutions proposed in this thesis.

The rest of the thesis is organized as follows. The related work is surveyed in Chapter 2. An overview of mixed fragmentation methodology is presented in Chapter 3. A representation scheme for mixed fragments and algorithms for merging the grid cells to mixed fragments is presented in Chapter 4. The algorithms to materialize distributed databases are presented in Chapters 5 and 6. A simulation model to estimate average transaction response time and a results of set of experiments conducted are presented in Chapter 7. A distributed database redesign methodology is presented in Chapter 8. A case study to demonstrate the viability of this methodology is presented in Chapter 9. Finally, conclusions and future research are presented in Chapter 10.

CHAPTER 2

RELATED WORK

2.1 General Comments

In this chapter a survey of the relevant work in the areas of distributed database design, performance analysis of distributed transaction processing and applications of Markovian decision analysis is presented. The solution to the redesign problem requires techniques from the above areas needs to be combined to develop an integrated methodology.

2.2 Distributed database design

The survey of work in developing fragmentation, allocation, redesign, materialization algorithms for distributed database design is presented first.

2.2.1 Fragmentation

Data fragmentation is performed during the design of a distributed database to improve the performance of the transactions. In order to improve transaction processing performance, fragments must be "closely matched" to the requirements of the transactions. In the design of a multi-site distributed database, fragments are allocated, and possibly replicated, at the various sites.

The work related to this topic can be classified into work on vertical fragmentation, horizontal fragmentation, and other fragmentation techniques.

Vertical Partitioning: Hoffer and Severance [HS75] measure the affinity between pairs of attributes and try to cluster attributes according to their pairwise affinity by using the bond energy algorithm (BEA) developed in [MSW72].

Navathe et al. [NCWD84] extend the results of Hoffer and Severance and propose a two phase approach for vertical partitioning. During the first phase, they use the given input parameters in the form of an attribute usage matrix and transactions, to construct the attribute affinity matrix on which clustering is performed. They cluster the attribute

affinity matrix by using a Bond Energy Algorithm [HS75]. This algorithm clusters large values of the matrix along the sides of the diagonal. This gives rise to partitioning the matrix into sub-matrices along the diagonal so that the values within the sub-matrix are more homogeneous than between sub-matrices. This partitioning is done by using an empirical objective function, the partitioning depends on the characteristics of this function. The sub-matrices generated by first level partitioning can be again partitioned to generate partitions at a finer level of detail. This iterative binary partitioning stops at the discretion of the designer.

Cornell and Yu [CY87] apply the work of [NCWD84] to relational databases. They propose an algorithm which decreases the number of disk accesses to obtain an optimal binary partitioning. They show how the knowledge of specific physical factors can be incorporated into the overall fragmentation methodology.

Ceri, Pernici and Wiederhold [CPW88] extend the work of [NCWD84] by considering it as a DIVIDE tool and by adding a CONQUER tool. Their CONQUER tool again extends the same basic approach in the direction of adding details about operations and physical accesses similar to [CY87]. This approach focuses on the decomposition of the design process into several design subproblems and provides no algorithmic improvement to the process of vertical partitioning itself.

Navathe and Ra [NR89], developed a graph-theoretic algorithm based on linearly-connected trees, and possibility of cycles, to detect all the vertical fragments of the relation in one iteration.

Chu and Ieong [CI91], develop a vertical partition method that optimizes the number of disk accesses based on clustering of attributes accessed by transactions, then they compare their method with other methods.

Chu [Chu92] takes a transaction oriented view of partitioning the attributes of a relation. He presents the concepts of *sufficient* and *support* to design the algorithms to vertically partition a relation. He also considers the problem of choosing the access path for the partitions generated.

Muthuraj et al. [MCVN93] address the problem of n-ary vertical partitioning by deriving a general purpose objective function based on achieving "ideal" clustering that corresponds to a minimal cost. This objective function is used for developing heuristic algorithms that are shown to satisfy the objective function. This objective function is also used to compare the previously proposed algorithms.

Horizontal Partitioning: Ceri, Negri and Pelagatti [CNP82] analyze the horizontal partitioning problem, dealing with the specification of partitioning predicates and the application of horizontal partitioning to different database design problems.

Ceri, Navathe and Wiederhold [CNW83] is the first definitive work on distribution design. In this paper they chose to develop an optimization model for horizontal partitioning without replication in the form of a linear integer zero-one programming problem. They proposed that horizontal partitioning may be addressed independently from vertical partitioning reducing the overall complexity of the distribution design process.

Yu et al. [YSLS85] propose an adaptive algorithm for record clustering, which is conceptually simple and highly intuitive. However, the adaptive approach does not use the given transaction information that is useful for partitioning, thus it cannot be used for the optimal design of distributed databases.

Shin and Irani [SI91] make use of the knowledge about the data stored in the relations along with user queries to infer the predicates defining the horizontal fragments. A many-sorted language has been extended to represent the queries and knowledge about the data stored in the relations compatibly.

Other Partitioning Techniques Apers [Ape88] considers the fragmentation problem together with the allocation problem. In his approach, the fragmentation scheme is the output of the allocation algorithm. Thus the fragmentation scheme is not known before the allocation. The emphasis of Aper's work is on allocation with fragmentation as a by product of the design.

2.2.2 Allocation

The allocation problem has been first treated in terms of file allocation problem in a multi computer system, and later on as a fragment allocation problem in distributed database system. Note that file allocation problem does not take into consideration the semantics of the processing being done on the files, whereas fragment allocation problem must take into consideration this aspect.

[Chu69] was the first to study the problem of file allocation with respect to multiple files on a multi processor system. He presented a global optimization model to minimize overall processing costs under the constraints of response time and storage capacity with a fixed number of copies of each file. Casey [Cas72] distinguished between updates and queries on files. Eswaran [Esw74] proved that Casey's formulation was NP complete. He suggested

that a heuristic rather than an exhaustive search approach is more suitable.

Ramamoorthy and Wah [RW83] analyzed a file allocation problem in the environment of a distributed database. Wah developed a heuristic approximation algorithm for the simple file allocation problem and the generalized file allocation problem. They had also proposed a model for file migration or reallocation that is identical in formulation to the file allocation problem.

Ceri, Martella and Pelagatti [CMP80] consider the problem of file allocation for typical distributed database applications with a simple model of transaction execution and without considering horizontal partitioning.

Apers [Ape88] considers the allocation of the distributed database to the sites so as to minimize the total data transfer cost. The author devised a very complicated scheme to allocate the relations by first partitioning them into an innumerable number of fragments, and then allocating them. The author integrated the system dependent query processing strategy with the logical model for allocating the fragments.

Gavish and Pirkul [GP82, GP86] address the combined problem of data and computer location. They basically develop an optimization model with a number of parameters, whose solution gives the location of both the data and computers. In this model it is very difficult to incorporate the query processing strategies of the distributed database management system and the semantics of the applications accessing the data. Even if this were included the resulting optimization model will have large number of variables, non-linearity, and can solve only small sized problems.

2.3 Redesign of distributed databases

In this section the preliminary work done in redesign and materialization of distributed databases is presented.

Redesign: Navathe and Wilson [WN86] classify the redesign problem into limited redesign and total redesign. They provide an iterative method to reallocate the fragments based on the change in sites from which the transactions are initiated. Other than this piece of work, there has been no work in this area to the best of our knowledge.

Materialization Rivera-Vega [VRVN89, RVVN90] worked on the problem of efficiently scheduling the data/file transfer operations among nodes of a network so as to redistribute data for a limited distributed database redesign. Other than this there has been no work

done on the materialization of distributed database in case of total redesign.

2.4 Response time analysis

The model for analyzing the distributed transaction response time proposed by P. S. Yu, et al;. [YDR+85, YDR+87, YDCI87, DYB87, CDY90] at IBM T. J. Watson Research Center is used to simulate the distributed transaction response time. Yu et al. developed models to analyze the effect of various parameters on distributed IMS database environments. A major bottleneck in the database system is the lock manager and the concurrency control mechanism which is adapted by the database management system. They develop a mathematical model in which they represent the transaction response time as a sum of time spent on CPU, I/O and the waiting time due to contention for accessing the same data (i.e. contention for the locks). They estimate the time spent on CPU, I/O and contention by using the queuing model.

There has been work done on analytical studies of specific concurrency control schemes; but they are not applicable to our research goal because they do not consider the effect of data and resource contention. Moreover, Yu's work is applicable for a wide variety of concurrency schemes and environments, whereas the earlier work by [TSG85, TGS85], is applicable to centralized systems with optimal or pessimistic concurrency control. Detailed simulation studies are reported in [ACL87], but are based on a fixed multi-programming level (MPL), whereas in this thesis an open queuing network model with no fixed MPL is considered. Carey and Livny [CL88] also perform a simulation study of the distributed database concurrency control performance algorithms. Again, they do this study with a fixed MPL, and fixed percentage of local transactions (70%), whereas in this thesis a response time study in an open system with the complete range of 0 to 100% of local transactions, and a range of disk i/o access times, communication delay, and arrival rates is dealt with. This study is the most exhaustive study undertaken with respect to the response time analysis of distributed transactions.

2.5 Markovian Decision Analysis

Howard, in his PhD thesis [How60] developed the Markov decision analysis as a dynamic programming technique to generate an optimal policy that selects the best alternative state

change for a Markov process where each state change has many alternatives, and each alternative has a different benefit value. This technique had been used in industrial engineering and operations research. Recently this technique had been applied to computer systems related problems by Nicol and Reynaolds [NRJ90] in optimal dynamic remapping of data parallel computations, and by Shin, Krishna and Lee [SKL89] on optimal dynamic control of resources in a distributed system. The details of the above two papers are not presented as they do not deal with database system related problems. Note that both these papers deal with a dynamic decision process, which is essentially what is needed for the redesign process in the "application processing center" scenario.

2.6 Summary

It can be seen from the above that there is very little work done on the redesign of distributed relational databases problem. Most of the work done has been in the area of developing algorithms for generating the fragmentation and allocation algorithms. There is work done on classifying the redesign problem and materializing the new allocation scheme in the limited redesign case. Therefore, we believe that this dissertation is the first piece of work that provides an overall methodology for redesigning the distributed databases in the application processing center environment. The solution for this problem uses techniques developed in the areas of simulation and performance analysis, and Markov decision analysis. The relevant work has been surveyed in this chapter.

CHAPTER 3

OVERVIEW OF THE MIXED FRAGMENTATION METHODOLOGY

3.1 General Comments

In this section an overview of the mixed fragmentation methodology for the initial distributed database design is given. This methodology was developed in the distributed database design tool (D^3T) project at the University of Florida. The methodology consists of two phases namely: grid creation and grid optimization. The grid creation phase is described in this chapter and the grid optimization phase is described in the next chapter. This methodology will also be used in generating new distributed database designs.

In the next section different alternatives for designing distributed databases are described. In the subsequent sections a brief description of the mixed fragmentation methodology, and the vertical and horizontal fragmentation algorithms for grid creation are presented.

3.2 Alternative approaches for initial distributed database design

There are different alternative approaches that have been proposed for the initial design phase of the distributed database design process. Some aspects of this initial design namely: vertical and horizontal fragmentation has been researched before. In Figure 1 the comprehensive list of alternatives available for the initial design are illustrated. The word "grid" is used to indicate the partitioning of a single relation achieved by simultaneously applying both horizontal and vertical partitioning. Other terms in that figure are explained as follows:

fragmentation + allocation refers to constructing fragments as well as deciding where to allocate them as a single decision problem.

grid optimization refers to a "merging operation" with respect to grid cells which combines the elementary/minimal grid cells into larger ones for minimizing the transaction

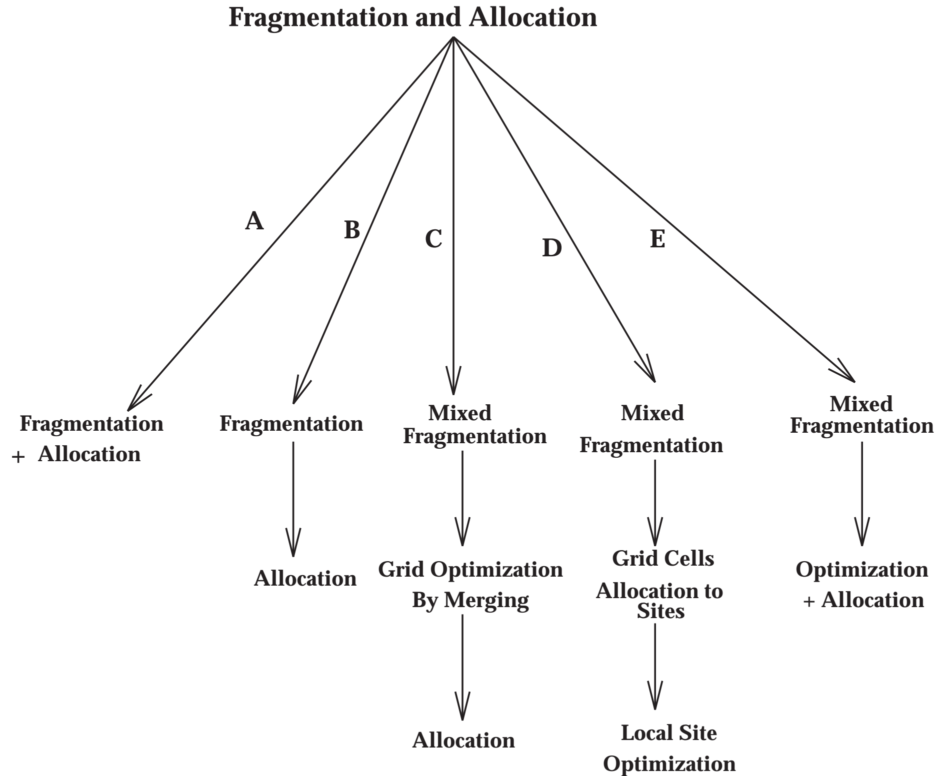


Figure 1: Alternatives for fragmentation and allocation

processing cost.

Our approach corresponds to the path C in this figure. Paths A and B leave vertical and horizontal partitioning as two separate problems as exemplified in [CNW83, NCWD84], the order in which they should be attacked is unspecified. Hence the resulting methodology is incomplete. Alternative A is a very hard problem to deal with since both fragmentation and allocation are simultaneously addressed. Alternatives C and D refer to the mixed fragmentation based on grid creation. The alternative C is the chosen approach because allocation is regarded as an entirely separate problem to be solved on the basis of much more detailed cost information. The fragmentation of a relation into grid cells and grid optimization are considered to be problems governed by the nature of data and its use by applications, whereas allocation deals with the problem of physical placement accounting for detailed transaction cost modeling. Alternative D is avoided again because it is more

complicated to deal with grid optimization and allocation together.

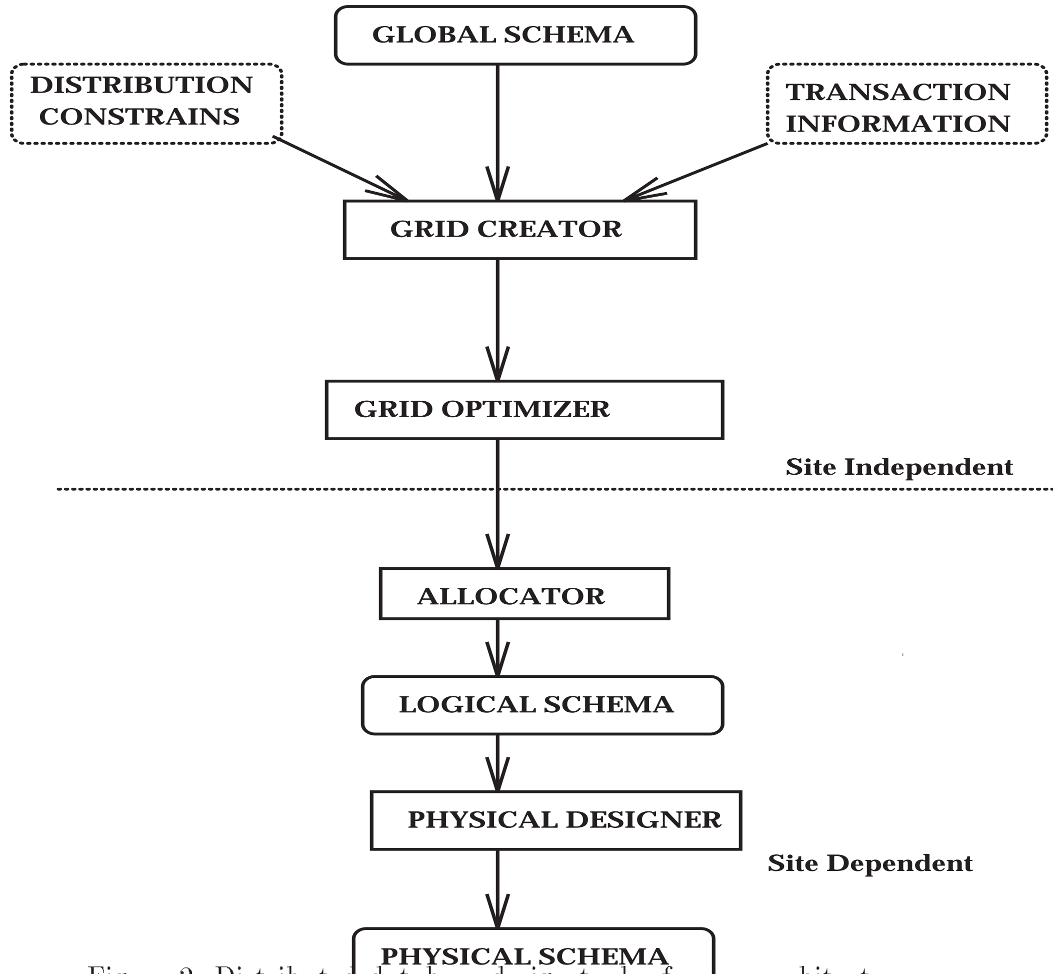


Figure 2: Distributed database design tool reference architecture

Figure 2 shows the Distributed Database Design Tool (D^3T) reference architecture which is used in our paper. The input information to D^3T is the global schema which consists of a set of relations, together with information about the important transactions on the proposed database. As reasoned in the previous work [CNW83, NCWD84], it is not necessary to collect information on 100% of the expected transactions (that would of course be impossible). Since the 80-20 rule applies to most practical situations, it is adequate to

supply the 20% of the heavily used transactions which account for about 80% of the activity against the database. The other input to the D^3T is the distribution constraints which include preferences or special considerations designers/users may have that would influence partitioning and allocation. In this paper we are going to deal with only the GRID CREATOR and GRID OPTIMIZER modules. The allocation problem has been considered in [Ra90] and is not included because of space constrains.

GRID CREATOR is composed of two modules; they are the vertical and the horizontal partitioning modules for grid creation. The output of the GRID CREATOR is a grid corresponding to a global relation. The grid suggests all possible ways in which the global relation in a distributed database may be partitioned. In this paper, each element of the grid is called a cell.

GRID OPTIMIZER After defining a grid, the GRID OPTIMIZER performs merging as much as possible according to the merging algorithms. The merging of the grid cells is an anti-fragmentation procedure. Having created the cells in a top-down fashion as the "minimal" fragments of a relation, it is considered whether the grid cells should be combined in a bottom-up fashion. Merging is considered desirable if it reduces the overall transaction processing cost. The representation scheme presented in Section 4.2 is used to develop an algorithm to generate the mixed fragments, the criteria for merging is to minimize the total number of disk accesses to execute the transactions developed in [CY87].

Note that in this thesis a single (global) relation mixed fragmentation dealt with, or one relation mixed fragmentation at a time is dealt with. The treatment of allocation should bring in the effect of processing several relations together by a transaction. Based on Figure 2, the proposed methodology for mixed fragmentation using a grid can be described as follows.

3.2.1 Mixed Fragmentation Methodology

This procedure involves all activities prior to allocation in the path C of Figure 1. The steps of the mixed partitioning methodology are illustrated in Figure 3. Note that the vertical and horizontal partitioning steps can be done concurrently.

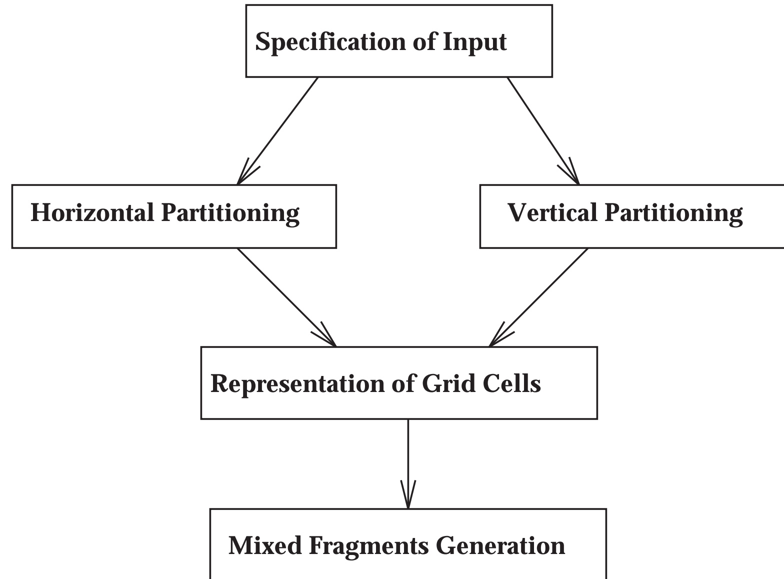


Figure 3: Steps in Mixed Fragmentation Methodology

1. *Specification of inputs:* In this step, the following inputs are specified.

   (a) *schema information:* relations, attributes, cardinalities, attribute sizes, predicates, etc.

   (b) *transaction information:* name, frequency, attribute usage, predicate usage, etc. The attribute usage matrix is a matrix containing transactions as rows and attributes as columns. Element $(i, j) = 1$ if transaction i uses attribute j, else it is 0. The transactions are also classified as "Retrieval" and "Update" types.

   (c) *distribution constraints:* any predetermined partitions or fixed allocation of data.

   (d) *system information:* number of sites, transmission costs, etc. This information is used particularly to solve the allocation problem.

   Figure 4 shows the transaction specification for the example being considered in this paper. The predicate descriptions are presented in the section 3.3.2.

2. *Vertical partitioning for grid:* In this step all candidate vertical fragments are determined. The graphical algorithm [NR89] is used for generating all fragments in one iteration. The overview of this algorithm is given in Section 3.1.

Transactions	Attributes	Predicates	Frequency	Sites of Origin
T_1	a_1, a_5, a_7	p_1, p_7	25	s_1, s_4
T_2	a_2, a_3, a_8, a_9	p_2, p_7	50	s_2, s_4
T_3	a_4, a_3, a_{10}	p_3, p_7	25	s_4
T_4	a_2,a_7,a_8	p_4, p_8	35	s_1
T_5	a_1,a_2, a_3, a_5, a_7, a_8, a_9	p_5, p_8	25	s_1, s_2, s_3
T_6	a_1, a_5	p_6, p_8	25	s_3, s_4
T_7	a_3, a_9	p_5, p_8	25	s_4
T_8	a_3, a_4, a_6, a_9, a_{10}	p_6, p_8	15	s_3, s_1, s_2

Figure 4: Transaction Specification

3. *Horizontal partitioning for grid:* In this step, all candidate horizontal fragments are determined. A method for this step is outlined in Section 3.3.2. Note that the sequence of steps 2 and 3 can be changed.

4. *Populating the system catalog with the representation of grid cells:* The representation scheme for the fragments and the grid cells will be used to generate the representation of the grid cells and to populate the system catalog. This representation scheme will be used by the algorithm to generate the mixed fragments.

5. *Mixed fragment generation:* The number of disk accesses required to execute a transaction will be used as a measure to come up with an optimal set of mixed fragments so as to minimize the total number of disk accesses required to process the transactions. This is a relevant measure which has been used in earlier studies [CY87, CI91, Chu92] to come up with an optimal set of vertical fragments. The concept of regular fragments with the cost measure of number of disk accesses develops a good solution to the mixed fragmentation problem.

3.3 Grid creation

3.3.1 Vertical partitioning for grid creation

Vertical partitioning is the process that divides a relation into sub-relations called vertical fragments containing groups of the original attributes [CNW83, CP84, NCWD84, MCVN93]. Most of the previous algorithms have started from constructing an attribute

affinity matrix from attribute usage matrix. Attribute affinity matrix is an $n \times n$ matrix for the n-attribute problem whose (i, j) element equals the "between-attributes" affinity that is the total number of accesses of transactions referencing both attributes i and j. An iterative binary partitioning method has been used [NCWD84, CY87, CPW88] based on first clustering the attributes and then applying empirical objective functions or mathematical cost functions to perform the fragmentation.

The graph theoretic algorithm presented in [NR89] starts from the attribute affinity matrix by considering it as a complete graph called the "affinity graph" in which an edge value represents the affinity between the two attributes, and then forms a linearly connected spanning tree. By a "linearly connected tree" that is, a tree that is constructed by including one edge at a time such that only edges at the "first" and the "last" node of the tree would be considered for inclusion. "Affinity cycles" in this spanning tree are formed by including the edges of high affinity value around the nodes and "growing" these cycles as large as possible. After the cycles are formed, partitions are easily generated by cutting the cycles apart along "cut-edges". Details of this algorithm as well as explanation of why it produces reasonable vertical fragments may be found in [NR89]. Henceforth for ease of reference the above algorithm is referred to as MAKE-PARTITION algorithm.

The major feature of this algorithm is that all fragments are generated by one iteration in a time of $O(n^2)$ that is more efficient than the previous approaches. This algorithm has been implemented and has been adapted it for the horizontal partitioning (Section 3.3.2).

	Attributes											
Transactions	1	2	3	4	5	6	7	8	9	10	Type	Access Freq
T_1	1	0	0	0	1	0	1	0	0	0	R	$acc_1 = 25$
T_2	0	1	1	0	0	0	0	1	1	0	R	$acc_2 = 50$
T_3	0	0	0	1	0	1	0	0	0	1	R	$acc_3 = 25$
T_4	0	1	0	0	0	0	1	1	0	0	R	$acc_4 = 35$
T_5	1	1	1	0	1	0	1	1	1	0	W	$acc_5 = 25$
T_6	1	0	0	0	1	0	0	0	0	0	W	$acc_6 = 25$
T_7	0	0	1	0	0	0	0	0	1	0	W	$acc_7 = 25$
T_8	0	0	1	1	0	1	0	0	1	1	W	$acc_8 = 15$

Figure 5: Attribute Usage Matrix

				Attributes						
Attributes	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	15	25	75	115	15
4	0	0	15	40	0	40	0	0	0	40
5	75	25	25	0	75	0	50	25	25	0
6	0	0	15	40	0	40	0	0	0	40
7	50	60	25	0	50	0	85	60	60	0
8	25	110	75	0	25	0	60	110	75	0
9	25	75	115	15	25	15	25	75	115	15
10	0	0	15	40	0	40	0	0	15	40

Figure 6: Attribute Affinity Matrix

Figure 5 shows the attribute usage matrix and Figure 6 shows an example of an attribute affinity matrix. Figure 7 shows the result of applying our algorithm to the attribute affinity matrix. In Figure 7 the nodes refer to attributes of the relation (i.e. node i refers to attribute a_i). The resulting vertical fragments are:

(1). (a_1, a_5, a_7)

(2). (a_2, a_3, a_8, a_9)

(3). (a_4, a_6, a_{10})

Here the major advantages of this method over the previous approaches are summarized:

1. There is no need for iterative binary partitioning. The major weakness of iterative binary partitioning used in [NCWD84] is that at each step two new problems are generated increasing the complexity; furthermore, termination of the algorithm is dependent on the discriminating power of the objective function.

2. The method obviates the need for using any empirical objective functions as in [NCWD84]. As shown by [CY87] the "intuitive" objective functions used in [NCWD84] do not necessarily work well when an actual detailed cost formulation for a specific system is utilized.

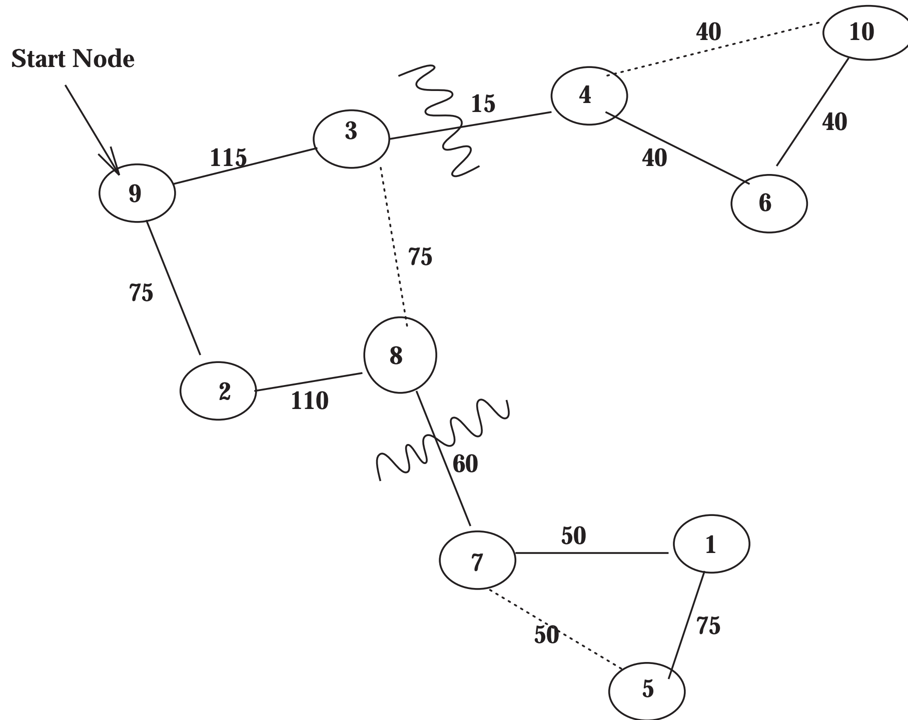


Figure 7: Vertical fragments generated by graph-theoretic algorithm

3. The method requires no complementary algorithms such as the SHIFT algorithm of [NCWD84].

4. The complexity of this approach is $O(n^2)$ as opposed to $O(n^2 logn)$ in [NCWD84].

In the following subsection it is shown how this algorithm may be applied to horizontal partitioning.

3.3.2 Horizontal partitioning for grid creation

Horizontal partitioning is the process that divides a global relation into subsets of tuples, called horizontal fragments [CNP82, CNW83, CP84]. Ceri, Negri and Pelagatti [CNP82] analyze the horizontal partitioning problem, dealing with the specification of partitioning predicates. Ceri, Navathe, and Wiederhold [CNW83] propose an optimization model for designing distributed database schemas with all meaningful candidate horizontal partitions. This approach developed an IP approach to select an optimal candidate horizontal partition for each relation.

In this thesis, however, the selection of an optimal horizontal partitioning for each relation in the database is not considered. But the focus is on identifying all the candidate

horizontal partitions. For this a horizontal partitioning methodology which will use an algorithm similar to the MAKE-PARTITION algorithm in [NR89] is proposed. In order to use the MAKE-PARTITION procedure of our vertical partitioning algorithm we consider only those transactions whose processing frequency is large (that is, those transactions that make up 80% of the database activity). These transactions access tuples of the relations based on some predicates. These predicates are specified as simple predicates [CNP82]. Another consideration in horizontal partitioning is that of derived partitioning as stated in [CP84]. The predicates which give rise to derived partitioning (we call it derived predicates) are considered in the same way as the simple predicates. The scope of this paper is limited by assuming that all simple and derived predicates are previously determined.

As explained earlier with respect to Figure 1, path C, the focus of this approach, is on single relation mixed fragmentation. Hence join predicates of the form R1.A=R2.B which deal with a pair of relations do not enter into the picture here. They are very much a part of the allocation phases and are motivated by minimizing the effort and cost of joins. In contrast, horizontal partitioning approaches used in systems like Bubba [CABK88] or Gamma [DGG+86] attempt to achieve parallelism of join queries.

The horizontal partitioning methodology is illustrated by using a simple example below. The inputs are a set of transactions and a corresponding set of predicates as follows (assume D-no and SAL are attributes of a relation):

1. T1 : D-no $<$ 10 (p1), SAL $>$ 40K (p7)
2. T2 : D-no $<$ 20 (p2), SAL $>$ 40K (p7)
3. T3 : D-no $>$ 20 (p3), SAL $>$ 40K (p7)
4. T4 : 30 $<$ D-no $<$ 50 (p4), SAL $<$ 40K (p8)
5. T5 : D-no $<$ 15 (p5), SAL $<$ 40K (p8)
6. T6 : D-no $>$ 40 (p6), SAL $<$ 40K (p8)
7. T7 : D-no $<$ 15 (p5), SAL $<$ 40K (p8)
8. T8 : D-no $>$ 40 (p6), SAL $<$ 40K (p8)

Note that the above set of predicates do not span all the tuples of the relation; tuples with SAL=40k will not be accessed by any of the above transactions.

The algorithm starts with the predicate usage matrix. The predicate usage matrix represents the use of predicates in important transactions. The predicate usage matrix is illustrated in Figure 8. Each row refers to one transaction where a "1" entry in a column indicates that the transaction uses the corresponding predicates. Whether the transaction uses the relation for retrievals or updates can also be captured by another column with R and U for retrievals and updates respectively.

	Predicates									
Transactions	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	Type	Access Freq
T_1	1	0	0	0	0	0	1	0	R	$acc_1 = 25$
T_2	0	1	0	0	0	0	1	0	R	$acc_2 = 50$
T_3	0	0	1	0	0	0	1	0	R	$acc_3 = 25$
T_4	0	0	0	1	0	0	0	1	R	$acc_4 = 35$
T_5	0	0	0	0	1	0	0	1	W	$acc_5 = 25$
T_6	0	0	0	0	0	1	0	1	W	$acc_6 = 25$
T_7	0	0	0	0	1	0	0	1	W	$acc_7 = 25$
T_8	0	0	0	0	0	1	0	1	W	$acc_8 = 15$

Figure 8: Predicate Usage Matrix

Predicate affinity is defined in a manner similar to attribute affinity [NCWD84]. Figure 9. shows a predicate affinity matrix generated from the predicate usage matrix in Figure 8. The numerical value of the (i, j) element in this matrix gives the combined frequency of all transactions accessing both predicates i and j and is obtained the same way as for vertical partitioning. The value $\Rightarrow$ of the (i, j) element indicates that predicate i implies predicate j, the value $\Leftarrow$ of the (i, j) element indicates that predicate j implies predicate i, and the value * represents the "close" usage of predicates. Two predicates i and j are "close" when the following conditions are satisfied:

1. i and j must be defined on the same attribute,
2. i and j must be jointly used with some common predicate c, and
3. c must be defined on an attribute other than the attribute used in i and j.

This is reasonable because predicates i, j, and c are different from one another and thus two fragments generated by predicates i, c and predicates j,c are considered "closely

related" since they both involve predicate c. In the above example, p_1 and p_2 are "close" because of their usage with common predicate p_7 in transaction T_1 and T_2. These two relationships are introduced to represent logical implication ($\Rightarrow$ and logical connectivity (*)) between predicates.

Predicates	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8
p_1		$\Rightarrow$, "*"	"*"	0	0	0	25	0
p_2	$\Leftarrow$, "*"		"*"	0	0	0	50	0
p_3	"*"	"*"		$\Leftarrow$	0	0	25	0
p_4	0	0	$\Rightarrow$		"*"	"*"	0	35
p_5	0	0	0	"*"		"*"	0	50
p_6	0	0	0	"*"	"*"		0	40
p_7	25	50	25	0	0	0		0
p_8	0	0	0	35	50	40	0	

Figure 9: Predicate Affinity Matrix

3.3.3 Procedure Horizontal-Partitioning

1. *Construct a predicate usage matrix:* Predicate usage matrix represents the use of predicates in important transactions. The predicate usage matrix for the example (8 predicates and 6 transactions) is shown in Figure 8. Each row refers one transaction: the "1" entry in a column indicates that the transaction uses the corresponding predicates.

2. *Form a predicate affinity matrix:* Predicate affinity is generated in a similar manner as attribute affinity. Figure 9 shows a predicate affinity matrix generated from the predicate usage matrix in Figure 8. The numerical value of the (i, j) element in this matrix gives the combined frequency of all transactions accessing both predicates i and j. The value "$\Rightarrow$" of the (i, j) element indicates that predicate i implies predicate j, the value "$\Leftarrow$" of the (i, j) element indicates that predicate j implies predicate i, and the value "*" means that two predicates i and j are "similar" in that both are used jointly with some predicate c. That is, transaction T_1 uses predicates i and c and transaction T_2 uses predicates j and c.

3. *Perform clustering of predicates:* It is done by using the modified version of the MAKE-PARTITION graphical algorithm in [NR89]. This modified algorithm is obtained by adding the following heuristic rules:

   (a) A numerical value (except zero) has higher priority than the values ⇒, "⇐" and "*" when selecting a next edge. This is because, more importance is placed on affinity values which are obtained from transaction usage rather than on logical connectivity among the predicates.

   (b) In comparisons involved in checking for the possibility of a cycle or extension of a cycle, cycle edges with affinity values "⇒", "⇐", and "*" are ignored. For example, in Figure 10, in comparing edge (p_4, p_8) with edges of the cycle (p_8, p_5, p_6) the edge (p_5, p_6) which has affinity "*" is ignored. This is because the affinity values "⇒", "⇐" and "*" represent implicit logical relationships among the predicates and not actual affinity between the predicates. These implicit relationships are used to reduce the number of horizontal fragments.

   (c) "⇒" and "⇐" are considered to have higher affinity value than "*" since the latter only represents logical connectivity between the two predicates through their usage with a common predicate.

   (d) If there are two "⇒" in a column corresponding to predicate p_k, one implied by predicate p_j, then the entry (i, k) has higher priority than the entry (j, k) , either if the entry (i, j) is equal to "⇐", or if the entry (i, j) is equal to "⇒". In other words, if $p_i \rightarrow p_j \rightarrow p_k$ then j has higher priority than i else $p_j \rightarrow p_i \rightarrow p_k$ then i has higher priority than j.

   In this step a set of subsets of predicates are obtained. In this example, by using the above rules and the MAKE-PARTITION algorithm, three subsets of predicates as shown in Figure 10: (p_1, p_7, p_2), (p_3, p_4), (p_5, p_6, p_8) are obtained.

4. *Optimize predicates in each subset:* In this step predicate inclusion and predicate implication are considered to minimize the number of predicates. In our example, the first subset (D-no $<$ 10, D-no $<$ 20, SAL$>$ 40K) is refined into (D-no $<$ 20, SAL$>$ 40K) since D-no $<$ 10 ⇒ D-no $<$ 20, the second one (D-no $>$ 20, 30$<$ D-no $<$ 50) is also refined into (D-no $>$ 20) since 30$<$ D-no $<$ 50 ⇒ D-no $>$ 20, but the last one (D-no $<$ 15, D-no $>$ 40, SAL$<$ 40K) has no change. Note that this optimization can

be done before step 3. But in this thesis, this step is performed here in order to allow pair of predicates such as p_1 and p_2 in which one (i.e. p_1) implies another (i.e. p_2) to be grouped in different clusters in step 3. The three clusters of predicates produced in this step called "cluster sets", are listed in Figure 10.

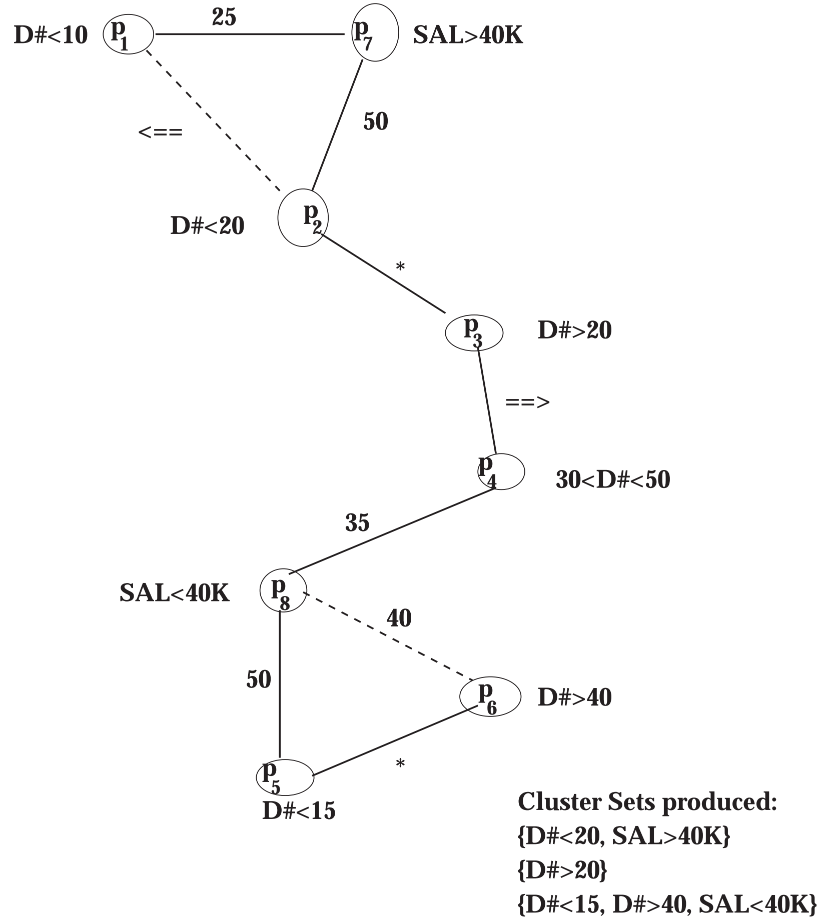


Figure 10: Clustering of predicates using graph-theoretic algorithm

5. *Compose predicate terms:* The cluster sets are first evaluated to determine "the least common attribute". In our example, since SAL does not appear in cluster set 2 (corresponding to the second cluster of predicates), it is the least common attribute. Note that D# appears in all three sets. A table called the "predicate term schematic table" is now considered by placing in the first column the chosen attribute with its appropriate ranges to cover that attribute exhaustively. In our example, two entries

are created: SAL $<$ 40 and SAL $>$ 40K for the SAL attribute. Then, the next to least common attribute is applied and then write its appropriate ranges that appear in the cluster sets against each entry for the first column. Note that these ranges may be overlapping. In our example, D# is the next attribute. Its ranges applicable to the cluster sets are: D# $<$ 15 or D# $>$ 40 coupled with SAL $<$ 40K (from cluster set 3), and D# $<$ 20 coupled with SAL $>$ 40K (from cluster set 1). The D# $>$ 20 predicate appearing in cluster set 2 must be written twice into the table against each entry for SAL. This resulting predicate term schematic table is shown at the top in Figure 11. Now predicate terms are constructed from the above tables as follows. Each horizontal entry in the table gives rise to one predicate term. If predicates refer to the same attributes then they are OR-ed (disjunction), otherwise they are AND-ed (conjunction). The resulting predicates are as follows:

(a) SAL$<$ 40K AND D-no $>$ 20,

(b) SAL$<$ 40K AND (D-no $<$ 15 OR D-no $>$ 40),

(c) SAL$>$ 40K AND D-no $<$ 20,

(d) SAL$>$ 40K AND D-no $>$ 20.

6. *Perform fragmentation:* There is one horizontal fragment per predicate term. Thus the number of horizontal fragments will at most be the number of predicate terms plus one because there is one remaining fragment which is the negation of the conjunction of all predicate terms.

7. *Restructure overlapping horizontal fragments:* The result of predicate partitioning may give rise to overlapping horizontal fragments. If merging of these fragments has to be considered, then the ADJUST function is needed to generate non-overlapping fragments. This should not affect the transaction processing because in case of vertical merging these horizontal fragments are again considered for merging to generate optimal horizontal or mixed fragments. This merging is done to minimize the transaction processing cost.

The resulting horizontal fragments are:

(a). (SAL $<$ 40k) AND (D# $>$ 20)

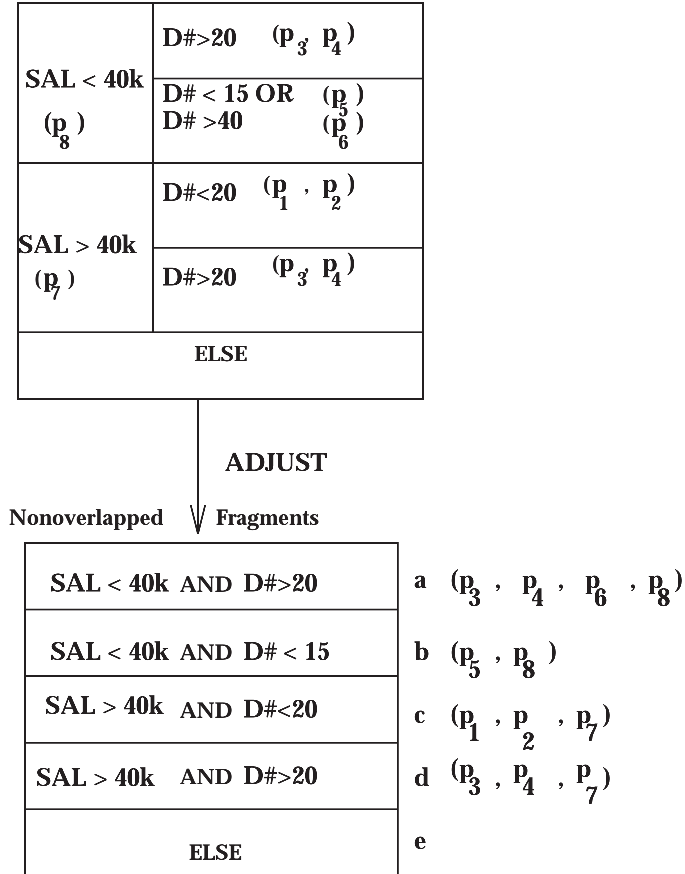


Figure 11: Non-overlapping horizontal fragments generation

(b). (SAL $<$ 40k) AND (D# $<$ 15)

(c). (SAL $>$ 40k) AND (D# $<$ 20)

(d). (SAL $>$ 40k) AND (D# $>$ 20)

(e). ELSE

The attractive features of this approach are as follows:

1. Fragments are based on actual predicates; by applying implication between predicates, the number of fragments is reduced,

2. The vertical and horizontal fragmentation are treated in a similar manner, and the algorithm for the vertical fragmentation with changes to incorporate ($\Rightarrow$, *) can be used for horizontal fragmentation.

3. 0-1 integer programming formulation is not needed, thus the complexity of the solution approach is reduced.

4. By using a clustering of predicates , a relatively small number of horizontal fragments are generated. The approach taken by [CNP82] can generate 2^n number of horizontal fragments for n simple predicates.

It should be noted that the complexity of this algorithm is dominated by the step 3, and thus will be $O(n^2)$ for n predicates as in the vertical partitioning algorithm in Section 3.3.1. A smaller value of n indicates a good understanding of the heavily used predicates by users. A comparable value for the number of predicates can be derived by clustering the tuples accessed by the transactions based on some attribute domain values, and defining these clusters as a predicate each. For example, if all transactions access the tuple based on a single attribute, then the attribute domain values accessed can be plotted on a real line, and clustered into sets, by using discriminant, each such cluster can be defined by a predicate. This approach can be extended to multi-dimensional case. Thus there is an approach to define the predicates by observing the attribute domain values used to access the tuples of the relation.

3.3.4 Grid Cells

In the last two subsections the algorithms for generating the vertical and horizontal fragmentation schemes have been described. Once the horizontal and vertical fragments are generated, the grid cells are generated by either applying the horizontal fragmentation scheme on each of the vertical fragments or by applying the vertical fragmentation scheme on each of the horizontal fragments. Therefore, each grid cell belongs to exactly one horizontal fragment and one vertical fragment. If the vertical fragmentation scheme generates n vertical fragments and the horizontal fragmentation scheme generates m horizontal fragments then $n \times m$ grid cells will be generated.

3.4 Summary

In this section a brief description of the grid creation phase of the mixed fragmentation methodology for the initial distributed database design tool has been given. The set of grid cells are generated by simultaneously applying the vertical and horizontal fragmentation scheme to a relation. The vertical fragmentation schemes are generated by using a common graph-theoretic algorithm `make_partition`. In [Ra90, NR89] the above algorithm is explained in detail. These algorithms alleviate the need of using complicated linear programming model for deriving horizontal fragments and iterative binary partitioning for deriving vertical fragments. These algorithms are efficient and generate all the candidate vertical and horizontal fragments in one iteration.

CHAPTER 4

REPRESENTATION SCHEME FOR MIXED FRAGMENTS

4.1 General Comments

In this section a representation scheme for the grid cells and the mixed fragments that are formed by merging the grid cells are developed. This representation scheme will be used in the materialization of redesign. The mixed fragmentation methodology facilitated the development of this representation scheme. This representation is used for developing algorithms to merge the grid cells into mixed fragments in this chapter. The procedure for merging the grid cells into a mixed fragment is based on the validity and correctness of the merging process and the resultant mixed fragments. This is because it should be possible to define the generated optimal mixed fragments as relations in relational databases without introducing null values. In the next section preliminary ideas on a representation scheme for grid cells are presented. After that a cost model for merging the grid cells and an algorithm to merge grid cells is developed.

4.2 Representation Scheme for Mixed Fragments

A grid is created by applying both the horizontal and vertical fragmentation schemes on the relation. Let $V = (1, 2, \ldots, n)$ be the set of vertical fragments and $H = (a, b, \ldots, x)$ be the set of horizontal fragments of the relation respectively. The grid generates a set of grid cells wherein each grid cell belongs to exactly one horizontal and one vertical fragment of the relation. The set of grid cells are represented as $(1_a, 1_b, \ldots, 1_x; 2_a, 2_b, \ldots, 2_x; \ldots; n_a, n_b, \ldots, n_x)$. For the examples presented in Sections 3.3.1 and 3.3.2, the vertical and horizontal fragmentation algorithms produced three vertical fragments and five horizontal fragments respectively. In Figure 12 the grid cells formed by simultaneously applying both these horizontal and vertical fragmentation schemes on the relation are represented.

The set of grid cells are classified as horizontal grid cells or vertical grid cells. The set of vertical fragments of a given horizontal fragment form *horizontal grid fragments*.

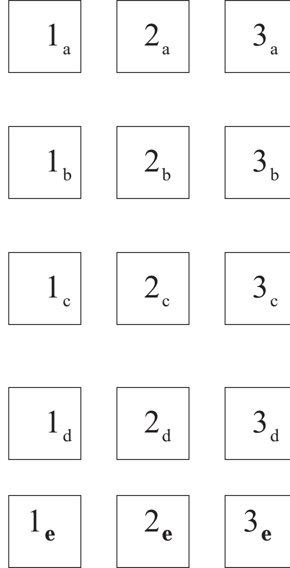


Figure 12: Representation of grid cells

For a horizontal fragment p they are represented as $(1_p, 2_p, \ldots, n_p)$. In Figure 12, the set $(1_b, 2_b, 3_b)$ forms the set of horizontal grid cells for the horizontal fragment b. The set of horizontal fragments of a vertical fragment are known as *vertical grid cells*. The vertical grid cells of a vertical fragment i are represented as $(i_a, i_b, \ldots, i_x)$. For example, in Figure 12 the set of grid cells $(2_a, 2_b, 2_c, 2_d, 2_e)$ form the vertical grid cells for the vertical fragment 2. Two binary operations **concatenate** ($\|$) *the horizontal merging operator* and **union** ($\bigcup$) *the vertical merging operator* on the set of horizontal grid cells and vertical grid cells respectively are defined. Concatenate operator is a special case of join operator where only corresponding tuple id's of the relations are matched. Note that all the relations involved in the concatenate operation have same number of tuples and that these relations have the same set of tuple identifiers.

Given two vertical grid cells i_p and i_q, the union of i_p and i_q is represented as $(i_{p,q}) = i_p \bigcup i_q$ and given two horizontal grid cells i_p and j_p the concatenation of i_p and j_p is represented as $(i_p, j_p) = i_p \| j_p$. The binary operations $\bigcup$ and $\|$ are commutative and associative over the set of vertical grid cells and horizontal grid cells respectively.

4.2.1 Characteristics of and valid operations on grid cells

The grid cells will be represented as α_β where $\alpha = 1, 2, \ldots, n$ and $\beta = a, b, \ldots, x$. α is the "column" index and β is the "row" index for a grid cell. A well formed expression over the set of grid cells is defined as follows:

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

1. $w = \alpha_\beta$, *is well formed where* $\alpha = 1, 2, \ldots, n$ *and* $\beta = a, b, \ldots, x$. $r(w) = \alpha_\beta$ *is the representation of* w.

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

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

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

   $w' = \alpha_\beta \bigcup w$ *is well formed if there exits a grid cell represented by* $\alpha_{\beta'}$ *in* $r(w)$. $r(w') = r(w) \bigcup \alpha_\beta$ *is its representation.*

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

This definition incorporates the discipline that is to be imposed while merging the grid cells during the grid optimization phase of the mixed fragmentation methodology. In Figure 12, the grid cells 1_a and 1_b can be merged (using the rule 2 above, since $\alpha_1 = \alpha_2 = 1$, and $\beta_1 =$ a, and $\beta_2 =$ b, implies $1_{(a,b)} = 1_a \bigcup 1_b$), whereas the grid cells 1_a and 2_c *cannot* be merged. Note that in forming the well formed expression, two grid cells are concatenated only if they both are horizontal grid cells of the same horizontal fragment, and a union of two grid cells is allowed only if they both are vertical grid cells of the same vertical fragment. Hence the conditions on the grid cells (α_β) in the rules 2 and 3 of the definition above. From the above definition of the well formed expression, a set of valid operations which need to be performed on the grid cells to generate a set of legal mixed fragments are derived. In general, a well formed expression w is represented as $(\alpha_{1A_1}, \alpha_{2A_2}, \ldots, \alpha_{pA_p})$ where $\alpha_j < \alpha_{j+1}$ for $1 \leq j \leq p-1$ with each α_i representing a vertical fragment and each A_i representing a set of vertical grid cells of vertical fragment α_i participating in the union.

Definition 2 *A* **regular well formed expression** *is a well formed expression* $w = (\alpha_{1A_1}, \alpha_{2A_2}, \ldots, \alpha_{pA_p})$ *such that* $A_1 = A_2 = \ldots = A_p$ *and* $\alpha_j = \alpha_{j-1} + 1$ *where* $2 \leq j \leq p$.

The first condition states that the set of vertical grid cells A_i participating in the formation of the fragment is the same for all vertical fragments α_i. The second condition states that all the vertical fragments α_i in the fragment are contiguous.

Definition 3 **Fragment** *is the result of a well formed expression over a set of grid cells. A* **regular fragment** *is the result of a regular well formed expression over the set of grid cells.*

For example, in Figure 12, the grid cells 1_a, 1_b, 2_a and 2_b form a regular fragment ($1_{(a,b)}$, $2_{(a,b)}$), whereas the fragment ($1_{(a,b)}$, 2_b) formed by merging the grid cells 1_a, 1_b, 2_b is not regular. It should be noted that each regular fragment corresponds to a table in the relational database. If the fragment is not a regular fragment, it will be difficult to represent it as a single relation without introducing null values.

Each grid cell belongs to exactly one horizontal and one vertical fragment and hence each grid cell is bounded by columns in the vertical fragment and the predicate defining the horizontal fragment it belongs to. A fragment has been defined as the result of a well formed expression over the set of grid cells. The attributes of a regular fragment are columns (given by the union of the columns of all vertical fragments in the well formed expression) of the fragment and the binding predicate (given by the disjunction of all the predicates defining the horizontal fragments in the well formed expression) of the fragment. All tuples in the fragment satisfy the binding predicate. The system catalog of the distributed database system needs to store and manage the metadata describing these attributes or characteristics of a fragment. The system catalog also stores and maintains the representation scheme along with the fragment names.

A transaction projects a set of attributes from a relation and selects tuples from the relation based on some conditions on the attributes of the relation defined by the predicates. Only those transactions that access single relations are considered. In case of transactions accessing multiple relations, there has to be either a join or union of relations. In case of union there is no condition spanning multiple relations, but in case of join there is at least one join condition that spans two relations. Since the join condition spans all the tuples of both relations that satisfy the condition imposed by other predicates pertaining

to the individual relations respectively. The join condition does not effect the horizontal partitioning schemes of the relations.

The span of a transaction t is defined as $\mathcal{S}$ $(t) = \{(C_1, C_2, \ldots, C_n); (P_1, P_2, \ldots, P_m)\}$, where $C_1, C_2, \ldots, C_n$ are columns being projected, and $P_1, P_2, \ldots, P_m$ are the predicates used in conditions to select the tuples being accessed by the transactions. Note that the partitioning the vertically based on attribute (column) affinities and horizontally based on predicate affinities means that the columns C_i's and predicates P_j's belong to some set of vertical fragments $\mathcal{C}$ $(t) = \{\alpha_t$ where $C_i \in \alpha_t, i = 1, 2, \ldots, n\}$ and horizontal fragments $\mathcal{P}$ $(t) = \{\beta_t$ where $P_j \in \beta_t, j = 1, 2, \ldots, m\}$ respectively.

Theorem 1 Transaction Accesses Regular Fragment
The set of grid cells $\{\alpha_{t\beta_t}$ *where* $\alpha_t \in$ $\mathcal{C}$ (t) *and* $\beta_t \in$ $\mathcal{P}$ $(t)\}$ *form a regular fragment.*

Proof: Let the set $\mathcal{P}$ (t) for a transaction t be $\{\beta_1^t, \beta_2^t, \beta_3^t, \ldots \beta_m^t\}$ = A, be the set of horizontal fragments accessed by the transaction t.

Let $\{\alpha_1^t, \alpha_2^t, \alpha_3^t, \ldots, \alpha_n^t\}$ be the set of vertical fragments accessed by transaction t.

Then for each of the vertical fragments α_i^t the transaction t accesses the tuples across the horizontal fragments A, because the predicates defining the conditions for filtering the tuples of the transaction t span the horizontal fragments in A.

Hence the transaction accesses all the grid cells defined by $\{\alpha_{1A}^t, \alpha_{2A}^t, \alpha_{3A}^t, \ldots, \alpha_{nA}^t\}$ and this is a representation of a regular fragment. Therefore, any transaction t accesses a set of grid cells representing a regular fragment. □

Definition 4 *Given* f_1 *and* f_2 *as two fragments with their representations* r_1 *and* r_2 *respectively. The intersection of two fragments,* f_1 **intersection** f_2 *as the fragment* $f_1 \bigcap f_2$ *whose representation is given by* $r(f_1 \bigcap f_2) = \{\alpha_\beta | \alpha_\beta \in r_1$ *and* $\alpha_\beta \in r_2\}$.

Theorem 2 Regular Fragment Intersection Closure *Intersection of two regular fragments is a regular fragment.*

Proof: Let f_1 and f_2 be two regular fragments with representations r_1 and r_2 respectively.

Let $r_1 = (\alpha_{1A_1}, \alpha_{2A_2}, \ldots, \alpha_{pA_p})$ where $A_1 = A_2 = \ldots = A_p$ and $\alpha_j = \alpha_{j-1} + 1$ where $2 \leq j \leq p$.

and

$r_2 = (\alpha'_{1_{A'_1}}, \alpha'_{2_{A'_2}}, \ldots, \alpha'_{q_{A'_q}})$ such that $A'_1 = A'_2 = \ldots = A'_q$ and $\alpha'_j = \alpha'_{j-1} + 1$ where $2 \leq j \leq q$.

If $r_1 \bigcap r_2$ is empty then it is a regular fragment.

Let $r_1 \bigcap r_2$ be not empty; then there exists α_β such that $\alpha_\beta \in r_1$ and $\alpha_\beta \in r_2$.

This implies that there exists some i and some j such that $\alpha_\beta \in \alpha_{i_{A_i}}$ and $\alpha_\beta \in \alpha'_{j_{A'_j}}$.

This implies that $A_i \bigcap A'_j \neq \emptyset$ which implies that $A_i \bigcap A'_j \neq \emptyset, \ \forall\, i, j$.

Let $B = A_i \bigcap A'_j$ for some i and j.

Let $I = \{\alpha_B \mid \alpha_B \in r_1 \bigcap r_2\}$. Let γ_1 be the smallest value of $\alpha \in I$ and

let γ_2 be the largest value of $\alpha \in I$.

Therefore $\gamma_1 \leq \gamma_2$. Then by the definition of fragments f_1 and f_2 it follows that $r(f_1 \bigcap f_2) = \{\gamma_{1B}, \ldots, \gamma_{kB}, \ldots, \gamma_{2B}\}$ where $\gamma_1 \leq k \leq \gamma_2$ (That is, $k \in \{\gamma_1, \gamma_1 + 1, \ldots, \gamma_2 - 1, \gamma_2\}$).

This representation is that of a regular fragment. Hence the intersection of two regular fragments is a regular fragment. □

4.2.2 Mapping of grid cells to transactions

The grid cells after the grid creation phase of the initial design are represented as $G = \{\alpha_\beta$ where $\alpha = 1, 2, \ldots, n$ and $\beta = a, b, \ldots, x\}$. The transactions accessing the grid cells are represented as $T = \{T_i$ where T_i accesses the relation $R\}$.

Let $\mathcal{T}$ be a function $\mathcal{T} : G \longrightarrow T$, be a function that maps the grid cells α_β to the transactions T_i which accesses them. Figure 13. shows the mapping of transactions to the grid cells for our example.

With respect to our example: $\mathcal{T}(1_a) = \{T_4, T_6\}$, $\mathcal{T}(1_b) = \{T_5\}$, $\mathcal{T}(1_c) = \{T_1\}$, $\mathcal{T}(2_a) = \{T_4, T_8\}$, $\mathcal{T}(2_b) = \{T_5, T_7\}$, $\mathcal{T}(2_c) = \{T_2\}$, $\mathcal{T}(3_a) = \{T_8\}$, $\mathcal{T}(3_d) = \{T_3\}$, and for the rest its $\emptyset$.

Define set $\mathcal{T}^{(1)} = \{\alpha_\beta \mid \mathcal{T}(\alpha_\beta) \neq \emptyset\}$.

Therefore, from the above example the set $\mathcal{T}^{(1)} = \{1_a, 1_b, 1_c, 2_a, 2_b, 2_c, 3_a, 3_d\}$.

Define $\mathcal{T}^{(2)} = \{(\alpha_{i_{\beta_i}}, \alpha_{j_{\beta_j}}) \mid \mathcal{T}(\alpha_{i_{\beta_i}}) \bigcap \mathcal{T}(\alpha_{j_{\beta_j}}) \neq \emptyset\}$.

This defines two grid cells $(\alpha_{i_{\beta_i}}, \alpha_{j_{\beta_j}})$ which are accessed by at least one transaction T_k. Therefore from the above example the set $\mathcal{T}^{(2)} = \{(1_a, 2_a), (2_a, 3_a), (1_b, 2_b)\}$.

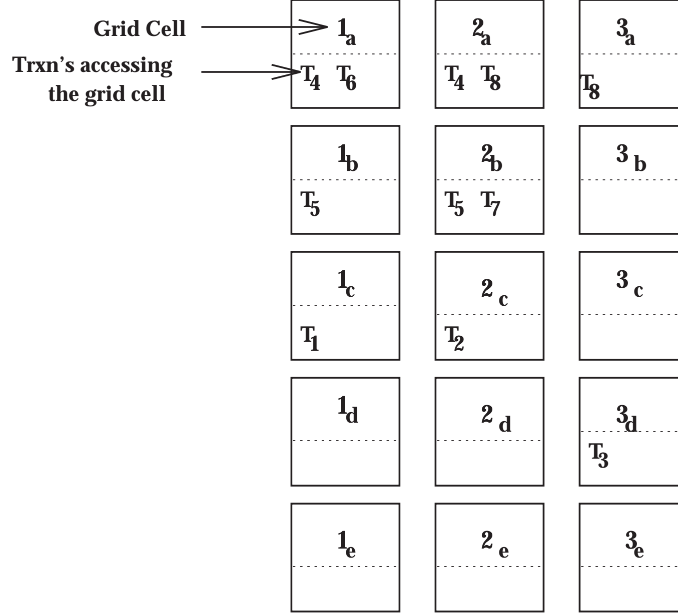


Figure 13: Mapping of Transactions to the Grid Cells

In general, define,

$\mathcal{T}^{(k)} = \{(\alpha_{i1_{\beta_{i_1}}}, \alpha_{i2_{\beta_{i_2}}}, \ldots \alpha_{i(k-1)_{\beta_{i(k-1)}}}, \alpha_{ik_{\beta_{i_k}}}) \mid \bigcap_{j=1}^{j=k} \mathcal{T}(\alpha_{ij_{\beta_{i_j}}}) \neq \emptyset\}$ where $k = 1, 2, \ldots, card(V) \times card(H)$, $\alpha_{ij} \in \{1, 2, \ldots n\}$ and $\beta_{ij} \in \{a, b, \ldots x\}$ and $\alpha_{ij_{\beta_{i_j}}}$'s are all distinct.

Represent

$\mathcal{T}^{(k)} = \{(f_1^k), (f_2^k), \ldots, (f_m^k)\}, k = 1, 2, \ldots, card(V) \times card(H)$

where each $f_i^k = (\alpha_{i1_{\beta_{i_1}}}, \alpha_{i2_{\beta_{i_2}}}, \ldots \alpha_{i(k-1)_{\beta_{i(k-1)}}}, \alpha_{ik_{\beta_{i_k}}})$.

With respect to the example above, $\mathcal{T}^{(2)} = \{f_1^2, f_2^2, f_3^2\}$ where $f_1^2 = (1_a, 2_a)$ and $f_2^2 = (2_a, 3_a)$ and $f_3^2 = (1_b, 2_b)$.

Moreover note that, $\mathcal{T}^{(3)} = \mathcal{T}^{(4)} = \ldots = \mathcal{T}^{(15)} = \emptyset$. This is because no transaction accesses more than two fragments. In general, there can be transactions accessing the complete relation.

4.3 Grid Optimization

In this section, a cost model to evaluate the merging decision is first described. After which the algorithms to merge the grid cells to optimal mixed fragments are presented.

4.3.1 Cost model for merging

Let the number of tuples (cardinality of the grid cell), the length of the tuple (in case there are variable length columns the average length of tuple) be known for each grid cell. The page size and the prefetch blocking factor are assumed to be the same constant across the distributed database environment and known in advance. Then the number of disk accesses taken by a transaction to access a grid cell by using segment scan is given by

$$Number\ of\ accesses = \frac{(Cardinality)(LengthOfTuple)}{((pagesize)(PreFetchBlockingFactor))}$$

The number of disk accesses required for a transaction by using an clustered index is:

$$Number\ of\ accesses = \frac{(Cardinality)(selectivity)(LengthOfTuple)}{(pagesize)}$$

The number of disk accesses required for a transaction by using an non-clustered index is

$$Number\ of\ accesses = (Cardinality)(Selectivity).$$

For a merged regular fragment represented by $\{\alpha_{1A}, \alpha_{2A}, \ldots, \alpha_{nA}\}$ its cardinality is the sum of the cardinalities of the vertical grid cells in A and the length of its tuple is the sum of the lengths of tuple[1] of the horizontal grid cells $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. But for the purposes of deciding whether to merge the grid cells or not, the above formulae are used. For a regular fragment f:

The number of disk accesses $\mathcal{N}_{\tau_s}(f)$ required by set of transactions τ_s to access the fragment using segment scan is given by:

$$\mathcal{N}_{\tau_s}(f) = \sum_{t \in \tau_s} \frac{(Cardinality(f_t))(LengthOfTuple(f_t))}{(pagesize)(prefetchblockingfactor)} freq_t$$

The number of disk accesses $\mathcal{N}_{\tau_c}(f)$ required by set of transactions τ_c to access the fragment using a clustered index is given by:

$$\mathcal{N}_{\tau_c}(f) = \sum_{t \in \tau_c} \frac{(Cardinality(f_t))(Selectivity(f_t))(LengthOfTuple(f_t))}{(pagesize)} freq_t$$

[1]Note that this is an approximate value that can be smaller or larger than the actual value depending on the implementation of the storage manager.

The number of disk accesses $N_{\tau_u}(f)$ required by set of transactions τ_u to access the fragment using a non-clustered index is given by:

$$\mathcal{N}_{\tau_u}(f) = \sum_{t \in \tau_u} (Cardinality(f_t))(Selectivity(f_t)) freq_t$$

Note that in the above formulae the $Cardinality$ and $LengthOfTuple$ values depend on the transaction t and the $freq_t$ denotes the frequency of executing the transaction t per unit interval of time.

Let $\mathcal{R}(f)$ denote a fragment f which is not merged (i.e. it is a loose collection of grid cells). Let f denote the merged fragment.

In order to decide whether or not to merge the grid cells to form a mixed fragment the total number of disk accesses required by all the transactions, using different indexing schemes and segment scan are considered. If the merging of grid cells results in a smaller number of disk accesses then the grid cells are merged else not.

That is, if:

$$\mathcal{N}_{\tau_s}(f) + \mathcal{N}_{\tau_c}(f) + \mathcal{N}_{\tau_u}(f) \leq \mathcal{N}_{\tau_s'}(\mathcal{R}(f)) + \mathcal{N}_{\tau_c'}(\mathcal{R}(f)) + \mathcal{N}_{\tau_u'}(\mathcal{R}(f))$$

then merge the grid cells. The CPU cost is ignored as it is very small compared to the disk I/O cost.

4.3.2 Merging algorithm

The motivation for merging grid cells is to increase the transaction processing efficiency, this is done by reducing the total number of disk accesses needed for processing all the transactions. A transaction can access one or more grid cells forming a regular fragment, also a grid cell can be accessed by one or more transactions. There are different types of accesses that can be used by the transactions namely, clustered index, non-clustered index and segment scan. Therefore only the sets of grid cells that are accessed by at least one transaction are considered. For each such set, we check whether they can be merged to form a *regular fragment*. If so, these sets of grid cells can be potentially merged.

The algorithm considers the merging possibilities in the sets of grid cells $f_i^k \in \mathcal{T}^{(k)}$ where $k = 2, 3, \ldots, card(V) \times card(H)$ and $j = 1, 2, \ldots, k_n$. Let $\mathcal{M}$ be the sets of grid cells that can be merged to form regular fragments. Then the algorithm for grid optimization is as follows:

Merging_Possibilities()

1. Let $\mathcal{M} = \emptyset$.

2. For $k = 2, 3, \ldots, card(V) \times card(H)$, consider $\mathcal{T}^{(k)}$:

   (a) For each set of grid cells (f_i^k) in $\mathcal{T}^{(k)}$ which forms a regular fragment, if

   $$\mathcal{N}_{\mathcal{T}_s}(f) + \mathcal{N}_{\mathcal{T}_c}(f) + \mathcal{N}_{\mathcal{T}_u}(f) \leq \mathcal{N}_{\mathcal{T}_s'}(\mathcal{R}(f)) + \mathcal{N}_{\mathcal{T}_c'}(\mathcal{R}(f)) + \mathcal{N}_{\mathcal{T}_u'}(\mathcal{R}(f)) \quad (1)$$

   then merge the grid cells else not.

   (b) If (f_i^k) is to be merged then $\mathcal{M} = \mathcal{M} \bigcup (f_i^k)$.

$\mathcal{M}$ is the set of sets of grid cells f_i^k which are possible candidates for merging. But some of these merging possibilities (f_i^k) may have some grid cells common with other merging possibilities.

Claim 1 *The algorithm* Merging_Possibilities() *generates all the mixed fragments.*

Justification: The above claim is supported by the following points:

1. The step 1. of the algorithm initializes the set of possible mixed fragments to an empty set.

2. The steps 2(a) and 2(b) are executed for each of the sets of grid cells defined by $(f_i^k) \in \mathcal{T}^{(k)}$ and the following points are validated:

   (a) The fact that the set of grid cells f_i^k form a regular fragment can be checked by using the definition of the *well formed expression*. If the representation of the well formed expression satisfies the condition for regular fragment, then the set of grid cells f_i^k forms a regular step and Equation 1 is tested.

   (b) For a set of grid cells f_i^k if equation 1 is satisfied then it implies that the number of disk I/O's is less if the set of grid cells f_i^k are merged to form a mixed fragment. The reason for this conclusion is presented in the Section 4.3.1.

   (c) In step 2(b), only those sets of grid cells f_i^k that reduce the number of disk I/O's to process all the transactions are included in the set M.

3. Therefore, the set $\mathcal{M}$ consisting of sets of grid cells f_i^k, all of which form mixed fragments. Since all the possible sets of grid cells f_i^k accessed by at least one transaction are tested to check if they form a regular fragment, and if the Equation 1 is satisfied for it to be included in set $\mathcal{M}$. The set $\mathcal{M}$ consists of all those sets of grid cells that form a mixed fragment, nothing more and nothing less. □

The above algorithm does generate an overlapping set of mixed fragments. That is, a set of mixed fragments wherein each grid cell may belong to one or more mixed fragments. Note that while forming the sets $\mathcal{T}^{(k)}$, a grid cell can belong to more than one sets of grid cells f_i^k, the mixed fragments generated by the above algorithm can be overlapping. Since mixed fragments so generated can be overlapping, the following two ways to decompose the overlapping mixed fragments into a set of non-overlapping mixed fragments are considered. The first case is the one in which each of the two mixed fragments have at least one grid cell which is not present in the other (overlapping case). The second case is where all grid cells belonging to one mixed fragment are contained in the other mixed fragment (contained-in case).

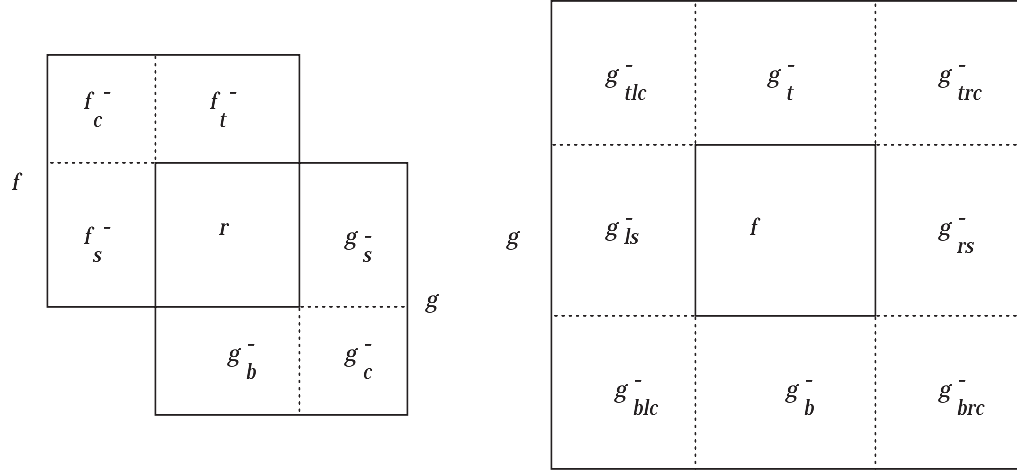
4.3.3 The case of overlapping fragments

Let f and g be two fragments that can be merged from the set $\mathcal{M}$ defined above, such that $f \bigcap g \neq \emptyset$. Note that f and g are both regular fragments. Let $r = f \bigcap g$, which is also a regular fragment. Define $f^- = f - r$ and $g^- = g - r$; note that $f^- \bigcap g^- = \emptyset$, $f^- \neq \emptyset$ and $g^- \neq \emptyset$. Now, in order to generate non-overlapping fragmentation scheme, r must be considered separately, and f^- or/and g^- must be considered separately.

Note that neither f^- nor g^- are regular fragments, but they can be defined as a collection of regular fragments, therefore f^- and g^- need to be represented as a collection of regular fragments. There may be more than one way to represent the fragments f^- and g^- as a collection of regular fragments. These alternatives are compared and one that minimizes the total number of disk I/O's to access the fragments f^- and g^- is selected.

The various alternatives for generating the non-overlapping mixed fragments from the overlapping mixed fragments are now described. As shown in the Figure 14, for two overlapping fragments f and g without any loss of generality[2] f can be written as, $f =$

[2]The operator $+$ is used as a common notation for Concatenation and Union operations so as not to



Overlapping Fragments

Contained-in fragments

Figure 14: Overlapping and Contained-in Fragments

$r + f_s^- + f_t^- + f_c^-$ and $g = r + g_s^- + g_b^- + g_c^-$, where f_s^- and g_s^- are the sets of grid cells to the sides of the regular fragment r, and f_t^- and g_b^- are the set of grid cells which are either above or below the regular fragment. And g_c^- and f_c^- are grid cells in the corner. Note that all of the above sets of grid fragments are regular and more over, $g_s^- \bigcup g_c^-$ is regular, $g_c^- \| g_b^-$ is regular, so is $f_s^- \bigcup f_c^-$, and $f_t^- \| f_c^-$.

In order to generate non-overlapping fragmentation schemes, the following alternatives for fragments f and g need to be considered.

Eval_Overlapping()

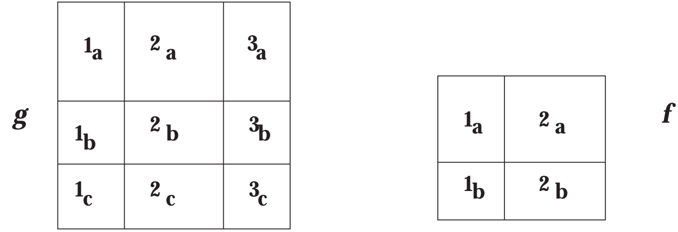
1. $\{f_s^-, f_t^-, f_c^-\}$ vs $\{f_s^-, f_t^- \| f_c^-\}$ vs $\{f_t^-, f_s^- \bigcup f_c^-\}$.

2. $\{g_s^-, g_b^-, g_c^-\}$ vs $\{g_s^-, g_c^- \| g_b^-\}$ vs $\{g_b^-, g_s^- \bigcup g_c^-\}$.

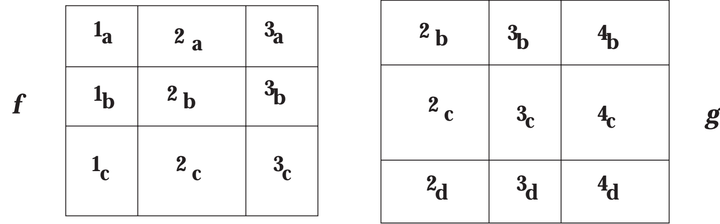
The alternatives, one each from (1) and (2) which give rise to least number of disk i/o's as described in section 4.3.1 are chosen. Each alternative redefines the fragment f^- or

complicate the expression. It is clear from the expression and Figure 14 as to when + means concatenation and when it means union.

g^- as a set of non-overlapping regular fragments. The regular fragments $\{f_s^-, f_t^-, f_c^-\}$ can be generated by comparing the representation scheme of fragment r with representation schemes of fragments f and g. The grid cells in f_t^- is defined by taking the grid cells in f with horizontal fragments not in r but vertical fragments in r. Similarly, the grid in f_s^- are defined by taking grid cells from f such that the horizontal fragments are the same as that of r, but vertical fragments are not those from r. And the grid cells in f_c^- are those from f such that none belong to either vertical or horizontal fragments from r. In a similar manner the grid cells in g_s^-, g_b^-, and g_c^- are defined. Note that in the above set of alternatives some of the elements like f_s^- may be empty sets.



Contained-in Grid Cells.



Overlapping Grid Cells

Figure 15: Example of overlapping and contained-in mixed fragments

With respect to example in Figure 15. for the overlapping mixed fragments f and g, $r = \{2_{(b,c)}, 3_{(b,c)}\}$, $f_t^- = \{2_a, 3_a\}$,$f_c^- = \{1_a\}$, $g_b^- = \{2_d, 3_d\}$, $g_c^- = \{4_d\}$, $f_s^- = \{1_{(b,c)}\}$, and $g_s^- = \{4_{(b,c)}\}$.

4.3.4 The case of contained in fragments

Let f and g be two fragments such that $f \bigcap g = f$. In this case fragment g can be written as, (see Figure 14) $g = (f + g_t^- + g_b^- + g_{ls}^- + g_{rs}^- + g_{blc}^- + g_{brc}^- + g_{tlc}^- + g_{trc}^-)^3$ where $g^- = g - f$. Note that the grid cells belonging to regular fragments g_{tlc}^-, g_t^-, etc,. are defined by ordering the horizontal fragments so that there are some above the horizontal fragments defining f, and some below, and vertical fragments to the left and right of those in f. Then using a similar reasoning as in the case of overlapping fragments, these regular fragments covering fragment f can be defined. For example, the regular fragment g_{trc}^- is defined by the grid cells generated by intersecting the set of horizontal fragments above those in f and a set of vertical fragments to the right of those in f. In order to generate non-overlapping fragments the following alternatives need to be considered. Note that g^- is not a regular fragment.

1. $\{g_{rs}^- \bigcup g_{trc}^- \bigcup g_{brc}^-, g_{ls}^- \bigcup g_{tlc}^- \bigcup g_{blc}^-, g_t^-, g_b^-\}$
2. $\{g_{rs}^- \bigcup g_{trc}^- \bigcup g_{brc}^-, g_{ls}^- \bigcup g_{tlc}^-, g_{blc}^-, g_t^-, g_b^-\}$ vs $\{g_{rs}^- \bigcup g_{trc}^- \bigcup g_{brc}^-, g_{ls}^- \bigcup g_{tlc}^-, g_{blc}^- || g_b^-, g_t^-\}$ vs $\{g_{rs}^- \bigcup g_{trc}^- \bigcup g_{brc}^-, g_{ls}^- \bigcup g_{blc}^-, g_{tlc}^-, g_t^-, g_b^-\}$ vs $\{g_{rs}^- \bigcup g_{trc}^- \bigcup g_{brc}^-, g_{ls}^- \bigcup g_{blc}^-, g_{tlc}^- || g_t^-, g_b^-\}$,
3. $\{g_{ls}^- \bigcup g_{tlc}^- \bigcup g_{blc}^-, g_{rs}^- \bigcup g_{trc}^-, g_{brc}^-, g_t^-, g_b^-\}$ vs $\{g_{ls}^- \bigcup g_{tlc}^- \bigcup g_{blc}^-, g_{rs}^- \bigcup g_{trc}^-, g_{brc}^- || g_b^-, g_t^-\}$ vs $\{g_{ls}^- \bigcup g_{tlc}^- \bigcup g_{blc}^-, g_{rs}^- \bigcup g_{brc}^-, g_{trc}^-, g_t^-, g_b^-\}$ vs $\{g_{ls}^- \bigcup g_{tlc}^- \bigcup g_{blc}^-, g_{rs}^- \bigcup g_{brc}^-, g_{trc}^- || g_t^-, g_b^-\}$,

Among all of the above choices the representation that has the least number of disk i/o's is chosen as described in section 4.3.1. Note that some of the fragments like g_{trc}^-, g_t^-, g_{ls}^-, g_{rs}^-, g_{trc}^-, g_{tlc}^-, g_{blc}^-, g_{brc}^- may be empty, thus reducing the number of alternatives.

With respect to example in Figure 15, $r = \{1_{(a,b)}, 2_{(a,b)}\}$, $g_t^- = g_{ls}^- = g_{blc}^- = g_{tlc}^- = g_{trc}^- = \emptyset$, $g_b^- = \{1_c, 2_c\}$, $g_{brc}^- = \{3_c\}$ and $g_{rs}^- = \{3_{(a,b)}\}$.

In this section an algorithm to generate the overlapping mixed fragments by merging the grid cells is presented. Two ways of decomposing the overlapping mixed fragments to non-overlapping mixed fragments are described. These algorithms are used to merge the sets of grid cells based on the transaction semantics and number of disk accesses required to process the transactions.

[3]The subscripts $_t$ and $_b$ mean top and bottom, $_{rs}$ and $_{ls}$ mean right and left side respectively and $_{blc}$, $_{brc}$, $_{tlc}$, $_{trc}$ represent the fragments at four corners of f, for e.g, top-left-corner, etc,.

Grid_Optimization()

1. Merging_Possibilities()
2. Sort the candidate mixed fragments in $\mathcal{M}$ in the descending order of the number of grid cells forming the mixed fragment.
   (a) Get the first mixed fragment f in $\mathcal{M}$ which has some other mixed fragment g, such that g is contained in f.
      i. Eval_Contained_in()
   (b) Else get the first mixed fragment f in $\mathcal{M}$ which has some other mixed fragment g, such that f and g are overlapping.
      i. Eval_Overlapping()
   (c) If there are none which are overlapping or contained-in, then exit else go to step 2.

Claim 2 *The algorithm* Grid_Optimization() *generates the non-overlapping mixed fragmentation.*

Justification: In the first step the routine Merging_Possibilities() is called to generate a set of merging possibilities to form mixed fragments. But as elaborated earlier the merging possibilities may generate an overlapping fragmentation scheme. Therefore, the mixed fragments in the merging candidates are sorted in descending order of the number of grid cells forming the mixed fragment. This is because the larger the mixed fragment, the greater the probability that it is overlapping with some other mixed fragment, or some other mixed fragment is contained in this fragment. The ordered list is searched until two mixed fragments that are overlapping or contained are found. The contained-in case is first considered so as to redefine the larger mixed fragment for processing all the transactions efficiently. In case of the overlapping fragments only some grid cells are common to both the fragments and redefinition of the fragments generates fewer new fragments than the contained-in case. Here the large fragments that have some grid cells in common with other fragments are redefined so as to generate as many non-overlapping mixed fragments as possible. After the two contained-in or overlapping fragments are considered and the non-overlapping mixed fragments are generated all the candidate mixed fragments (including the newly generated ones) are again sorted. Note that in this step the non-overlapping fragments

are generated by selecting the alternative that requires the least number of disk I/O accesses to process all the transactions. This process of finding a pair of contained-in or overlapping fragments and redefining is iteratively continued till non-overlapping fragmentation scheme is generated. This set of final candidate mixed fragments gives us the mixed fragmentation scheme for the relation.

Therefore, the algorithm *Grid_Optimization()* generates a non-overlapping mixed fragmentation scheme. As the algorithm is a greedy, its optimality cannot be guaranteed. □

4.4 Summary

In this chapter a representation scheme for the mixed fragments was presented. A cost model on which our merging algorithms are based is presented. An algorithm that generates an overlapping mixed fragmentation scheme was proposed. In order to generate non-overlapping mixed fragments, the cases of overlapping mixed fragments namely, contained-in and overlapping fragments are considered. The algorithms to redefine the overlapping or contained-in fragments as non-overlapping mixed fragments are developed. Finally, an algorithm that generates the non-overlapping mixed fragmentation scheme is presented. The major feature of this methodology was that it incorporates horizontal and vertical fragmentation simultaneously based on the same algorithm and supports the investigation of the effects of the different sequences of partitioning. The top-down philosophy was used for generating grid cells corresponding to clusters of predicates and attributes accessed by transactions together. Then the "bottom-up" evaluation was done by considering a reduction in cost by merging the already created cells vertically or horizontally. This is the first comprehensive treatment for generating a mixed fragmentation scheme in distributed database design.

CHAPTER 5

MATERIALIZATION OF DISTRIBUTED DATABASES

5.1 General Comments

This chapter presents a methodology to materialize a new design of a populated distributed database from its existing current design. It is assumed that both vertical and horizontal partitioning are simultaneously applied to generate a set of grid cells of the relation. A representation scheme developed in the last chapter for defining the fragments as a well formed expression over the grid cells will be used in developing the materialization algorithms. Two approaches shall be presented to materialize the distributed databases namely: *the query generator approach* and the *operator approach*. For both the approaches the algorithms are presented and their correctness is proved. In developing these algorithms to materialize the distributed database, the case where the grid cells do not change is considered first. That is, there is no change in the vertical and horizontal fragmentation schemes between the current design and the new design. After that the solution is extended to the case where there is a change in the grid cells. A cost model was developed to evaluate the cost of materializing the distributed database. This model was used to compare the two approaches to materialize the distributed database.

5.2 Two approaches to materialize a new design

The distributed database design (F, A) consists of a fragmentation (F) and an allocation (A) scheme. The total redesign generates a new fragmentation and allocation scheme from the current design represented by (F', A'). The objective is to develop a methodology and a set of algorithms to materialize (F', A') from (F, A) for a populated distributed database. The following two approaches have been developed:

Query Generator In this approach each of the fragments in the new fragmentation scheme is expressed as a query on the distributed database so as to let the distributed database system materialize the new fragmentation scheme. This approach depends on the

ability to generate correct queries for defining each of the fragments. The distributed database system undertakes the task of materialization of the new design. The module which generates the above set of statements is known as the **query generator**. Developing a query generator and proving its correctness is a complicated problem. An algorithm for query generation is presented in Section 5.4.1.

Operator Method The motivation for this approach is that the query generator method depends on the processing capability of the distributed database system and it may become unwieldy and out of control to give optimal performance. The operator method consists of defining a set of primitive operations which can be performed on the fragments. The operations are *"split", "move", "replicate", "remove" and "merge"*. This method was utilized in limited redesign by Wilson and Navathe [WN86]. The operation split is used to split a fragment into two sub-fragments. The operations move, replicate and remove are used to move a fragment from one site to another site, replicate a given fragment at another site, and delete a fragment at some site respectively. The operation merge is used to form a single fragment by merging two fragments. The operator method generates a program with above mentioned operations which when executed will materialize the new design. The complexity of these algorithms depends on the complexity of the underlying split, merge and move operations. The module supporting these operations is external to the distributed database system. The algorithms to support these operations are subject to further analysis so that they can be efficiently implemented.

In case of the query generator method, the efficiency of the method depends on the efficiency of the distributed database system and is typically beyond the control of the distributed database administrator.

5.3 Representation of Distribution Designs

A formal definition of the fragmentation scheme and a few other concepts that will be used in developing the algorithms to materialize the redesigned distributed databases are presented first.

Definition 5 Fragmentation scheme *is a set of* regular fragments *over the set of grid cells such that each grid cell belongs to* **at least one** *fragment.*

Definition 6 Non-overlapping fragmentation scheme *is a fragmentation scheme such that each of the grid cells belongs to* **exactly one** *fragment of the fragmentation scheme.*

Consider the following example of the employee relation shown in Figure 16 with vertical fragments as:

EMPLOYEE								
Emp#	**Name**	**City**	**Dept#**	**Proj#**	**Salary**	**Bonus**	**School**	**Degree**
1	Ian	Atl	10	123	20K	1K	GaTech	BS
2	Jim	SF	11	124	50K	2K	UnivFL	MS
3	Tom	LA	2	231	30K	1K	Duke	BS
4	Ann	NY	3	231	65K	5K	CalTech	MS
5	Kate	NO	21	123	33K	2K	UCB	BS
6	John	LDN	22	145	110K	8K	GaTech	Ph.D.
7	Ram	BOM	13	165	68K	6K	IIT	M.Tech
8	Mary	STL	33	213	45K	3K	NYU	BS
9	Vijay	DEL	33	213	40K	2K	GaTech	BS

Figure 16: Relation Employee

1. {Emp#, Name , City},

2. {Emp#, Dept#, Proj#},

3. { Emp#, Salary, Bonus},

4. {Emp#, School, Degree}.

and horizontal fragments as:

a. { Dept# $<$ 10},

b. { 10 $\leq$ Dept# $<$ 20},

c. { $20 \leq$ Dept# < 30},

d. { Dept# ≥ 30 }.

Figure 17 illustrates the 16 grid cells generated by simultaneously applying the vertical fragmentation (which derives four vertical fragments 1, 2, 3 and 4) and horizontal fragmentation (which derives four horizontal fragments a, b, c and d).

Emp#	Name	City
3	Tom	LA
4	Ann	NY

Emp#	Dept#	Proj#
3	2	231
4	3	231

Emp#	Salary	Bonus
3	30k	1k
4	65k	5k

Emp#	School	Degree
3	Duke	BS
4	CalTech	MS

Emp#	Name	City
1	Ian	Atl
2	Jim	SF
7	Ram	BOM

Emp#	Dept#	Proj#
1	10	123
2	11	124
7	13	165

Emp#	Salary	Bonus
1	20k	1k
2	50k	2k
7	68k	6k

Emp#	School	Degree
1	GaTech	BS
2	UnivFL	MS
7	IIT	M.Tech

Emp#	Name	City
5	Kate	NO
6	John	LDN

Emp#	Dept#	Proj#
5	21	123
6	22	145

Emp#	Salary	Bonus
5	33k	2k
6	110k	8k

Emp#	School	Degree
5	UCB	BS
6	GaTech	Ph.D.

Emp#	Name	City
8	Mary	STL
9	Vijay	DEL

Emp#	Dept#	Proj#
8	33	213
9	33	213

Emp#	Salary	Bonus
8	45k	3k
9	40k	2k

Emp#	School	Degree
8	NYU	BS
9	GaTech	BS

Figure 17: Grid Cells of the Relation Employee

Figure 18 shows the result of the grid optimization phase on the grid cells in Figure 17. This results in four fragments $DEPT_PROJ$, $SAL_SCHOOL1$, $SAL_SCHOOL2$ and $SAL_SCHOOL3$.

Figure 19 shows the representation of the fragmentation scheme (F, A) (illustrated in Figure 18) with four fragments f_1, f_2, f_3 and f_4. For ease in development of general purpose algorithms to materialize redesigned distributed database the fragment names instead of specific table names will be used.

DEP_PROJ

Emp#	Name	City	Dept#	Proj#
1	Ian	Atl	10	123
2	Jim	SF	11	124
3	Tom	LA	2	231
4	Ann	NY	3	231
7	Ram	BOM	13	165
5	Kate	NO	21	123
6	John	LDN	22	145
8	Mary	STL	33	213
9	Vijay	DEL	33	213

SAL_SCHOOL1

Emp#	Salary	Bonus	School	Degree
1	20k	1k	GaTech	BS
2	50k	2k	UnivFL	MS
3	30k	1k	Duke	BS
4	65k	5k	CalTech	MS
7	68k	6k	IIT	M.Tech

SAL_SCHOOL2

Emp#	Salary	Bonus	School	Degree
8	45k	3k	NYU	BS
9	40k	2k	GaTech	BS

SAL_SCHOOL3

Emp#	Salary	Bonus	School	Degree
5	33k	2k	UCB	BS
6	110k	8k	GaTech	Ph.D.

Figure 18: Current Design (F, A)

Table 1: Representation of Current Design (F, A).

Fragment Name	Table Name	Representation	Site
f_1	DEPT_PROJ	$(1_{(a,b,c,d)}, 2_{(a,b,c,d)})$	s_1
f_2	SAL_SCHOOL1	$(3_{(a,b)}, 4_{(a,b)})$	s_2
f_3	SAL_SCHOOL2	$(3_{(c)}, 4_{(c)})$	s_1
f_4	SAL_SCHOOL3	$(3_{(d)}, 4_{(d)})$	s_3

Each grid cell belongs to exactly one horizontal and one vertical fragment and hence each grid cell is bounded by columns in the vertical fragment and the predicate defining

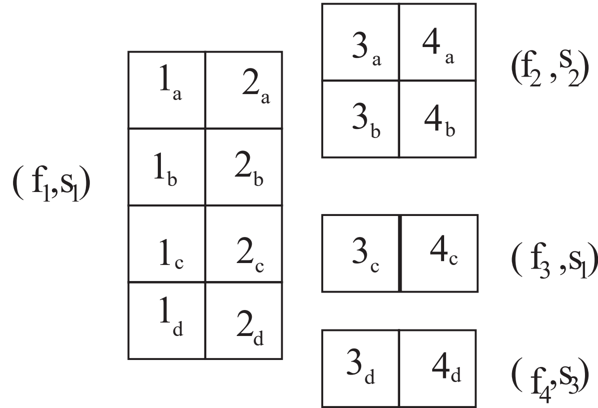


Figure 19: Current Fragmentation Scheme (F, A)

the horizontal fragment it belongs to. A fragment has been defined as the result of a well formed expression over the set of grid cells. The attributes of a regular fragment are columns (given by the union of the columns of all vertical fragments in the well formed expression) of the fragment and the binding predicate (given by the disjunction of all the predicates defining the horizontal fragments in the well formed expression) of the fragment. All the tuples in the fragment satisfy the binding predicate. The system catalog of the distributed database system needs to store and manage the meta data describing these attributes or characteristics of a fragment. The system catalog also stores and maintains the representation scheme along with the fragment names.

5.3.1 Intersection of fragmentation schemes

Let R be a relation, with $V = \{1, 2, \ldots, n\}$ as the set of vertical fragments, $H = \{a, b, \ldots, x\}$ be the set of horizontal fragments and $\{1_a, 1_b, \ldots, 1_x; 2_a, 2_b, \ldots, 2_x; \ldots; n_a, n_b, \ldots, n_x\}$ be the set of grid cells. Assume the current design (F, A) and the new design (F', A') for the relation R is given. Then, let $F = \{f_1, f_2, \ldots, f_n\}$ and $F^{'} = \{f_1^{'}, f_2^{'}, \ldots, f_m^{'}\}$ be two fragmentation schemes on R. As each fragment $f_i^{'}$ is a sub-fragment of R, R covers[1] $f_i^{'}$. It may be possible to cover a fragment of $f^{'}$ by using a subset of the fragments of F.

[1]By covers, it means that each tuple of the fragment $f_i^{'}$ is a tuple or part of a tuple of the relation R.

Definition 7 *The* **Minimal cover** *of a fragment $f_i^{'}$ of a fragmentation scheme $F^{'}$ is the set of fragments of fragmentation scheme F which have non-empty intersection with $f_i^{'}$. That is, $m(f_i^{'}) = \{f_j \mid r(f_j) \bigcap r(f_i^{'}) \neq \emptyset \text{ where } f_j \in F\}$.*

Thus the minimal cover of a fragment ($f^{'}$) of the new fragmentation scheme (F') consists of the fragments (f) from the current fragmentation scheme (F) containing data to form it. The theorem below guarantees that no spurious information is generated when materializing the new design.

Theorem 3 No Spurious Information Generation *There is no fragment in F' whose minimal cover is empty.*

Proof: Every grid cell of relation R belongs to some fragment f_i of F and $f_j^{'}$ of $F^{'}$; therefore, $r(f_i) \bigcap r(f_j^{'}) \neq \emptyset$. Hence the minimal cover of any fragment of $F^{'}$ is not empty. □

EDP

Emp#	Name	City	Dept#	Proj#
1	Ian	Atl	10	123
2	Jim	SF	11	124
3	Tom	LA	2	231
4	Ann	NY	3	231
7	Ram	BOM	13	165

ESB

Emp#	Salary	Bonus
1	20k	1k
2	50k	2k
3	30k	1k
4	65k	5k
7	68k	6k

ESD1

Emp#	School	Degree
1	GaTech	BS
2	UnivFL	MS
3	Duke	BS
4	CalTech	MS
7	IIT	M.Tech
5	UCB	BS
6	GaTech	Ph.D.

EDPS

Emp#	Name	City	Dept#	Proj#	Salary	Bonus
5	Kate	NO	21	123	33k	2k
6	John	LDN	22	145	110k	8k
8	Mary	STL	33	213	45k	3k
9	Vijay	DEL	33	213	40k	2k

ESD2

Emp#	School	Degree
8	NYU	BS
9	GaTech	BS

Figure 20: New Fragmentation Scheme (F', A')

The **intersection scheme** of fragmentation schemes F and $F^{'}$ is defined as $F \bigcap F^{'} = \{m(f_1^{'}), m(f_2^{'}), \ldots, m(f_m^{'})\}$. Regular Fragment Intersection Closure theorem assures that the set of fragments in $F \bigcap F'$ are all regular fragments, and that the fragments in the minimal cover are also regular fragments. The motivation for the intersection scheme is that for forming any fragment $f^{'}$ of the fragmentation scheme $F^{'}$, data from fragments $m(f^{'})$ of the fragmentation scheme F is needed. The cardinality of $m(f_i')$ gives the number of sub-fragments that need to be merged to form the fragment f'. This intersection scheme provides an approach for generating the merge operations on the fragments of the fragmentation scheme F. Also, it is proved that there is no fragment in fragmentation scheme F' whose minimal cover is empty. This means that each of the fragments of the new fragmentation scheme has a non-trivial minimal cover and that there is a mechanism to generate this minimal cover.

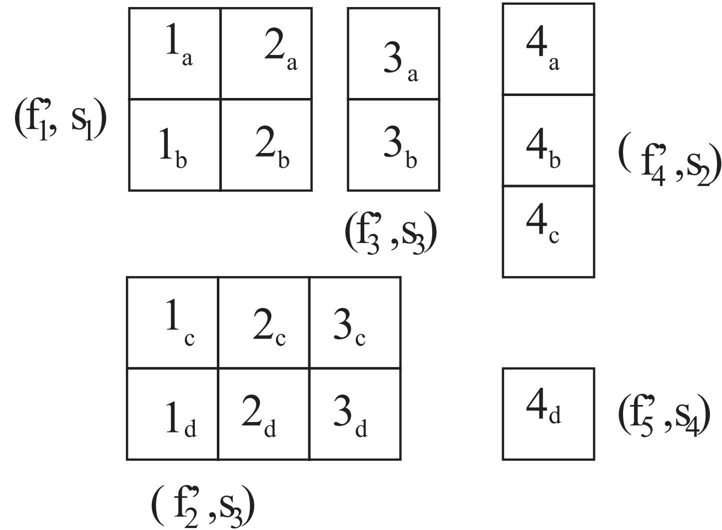


Figure 21: Representation of Fragmentation Scheme (F', A')

Figure 21 gives the representation of new design (F', A') of Employee relation given in Figure 16. The minimal covers for the fragments of the fragmentation scheme F are given in Table 3.

For example, from Table 3, to form the fragment f_2', data from the fragments f_1, f_3 and f_4 which are the elements of its minimal cover is needed.

Table 2: Representation of New Design (F', A').

Fragment Name	Table Name	Representation	Site
f_1'	EDP	$(1_{(a,b)}, 2_{(a,b)})$	s_1
f_2'	EDPS	$(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$	s_3
f_3'	ESB	$(3_{(a,b)})$	s_3
f_4'	ESD1	$(4_{(a,b,c)})$	s_2
f_5'	ESD2	$(4_{(d)})$	s_4

Table 3: Minimal Cover for Fragments in the New Design

New Fragment (f')	Elements (f) of Minimal Cover	Representation $r(f \bigcap f')$	Minimal Cover Site (From Current Allocation)
f_1'	f_1	$(1_{(a,b)}, 2_{(a,b)})$	s_1
f_2'	f_1	$(1_{(c,d)}, 2_{(c,d)})$	s_1
	f_3	(3_c)	s_1
	f_4	(3_d)	s_3
f_3'	f_2	$(3_{(a,b)})$	s_2
f_4'	f_2	$(4_{(a,b)})$	s_2
	f_3	(4_c)	s_1
f_5'	f_4	(4_d)	s_3

Definition 8 *An* **intersect** *of a fragment f_i of fragmentation scheme F is defined as the set of fragments of fragmentation scheme F' which intersect with the fragment f_i. That is, $int(f_i) = \{f_j' \mid r(f_i) \bigcap r(f_j') \neq \emptyset \text{ where } f_j' \in F'\}$.*

Thus the intersect of a fragment (f) of the current fragmentation scheme (F) consists of the fragments (f') of the new fragmentation scheme (F') that need data from it. The theorem below guarantees that there is no loss of information when the new design is materialized.

Theorem 4 No Loss of Information *There is no fragment in F whose intersect is empty.*

Proof: The proof is similar to that of theorem 2. □

Intuitively, the above theorem says that every grid cell in the old design must occur somewhere in the new design. Otherwise, some data is lost. The cardinality of the $int(f_i)$

of a fragment f_i gives the number of sub-fragments the fragment f_i is split into. In the above theorem it is proved that each fragment f_i has a non-empty intersect, therefore each fragment f_i needs to contribute its data to form some fragment f'_j of fragmentation scheme F'. Table 4 lists the intersect for each of the fragments of the fragmentation scheme F'.

Table 4: Intersect for Fragments in the Current Design

Old Fragment (f)	Elements (f') of Intersect	Representation $r(f \bigcap f')$	Intersect Site (New Allocation)
f_1	$f_1^{'}$ $f_2^{'}$	$(1_{(a,b)}, 2_{(a,b)})$ $(1_{(c,d)}, 2_{(c,d)})$	s_1 s_3
f_2	$f_3^{'}$ $f_4^{'}$	$(3_{(a,b)})$ $(4_{(a,b)})$	s_3 s_2
f_3	$f_2^{'}$ $f_4^{'}$	$(3_{(c)})$ $(4_{(c)})$	s_3 s_2
f_4	$f_2^{'}$ $f_5^{'}$	$(3_{(d)})$ $(4_{(d)})$	s_3 s_4

For example, Table 4 illustrates that the intersect of fragment f_2 consists of fragments f'_3 and f'_4. The above theorems shall be used in developing the algorithms for materializing the design using the operator method. Now the characteristics of a design are presented by combining a fragmentation scheme with an allocation scheme.

Let S be the set of sites in the distributed database environment. An allocation scheme A is a mapping between the set of fragments in the fragmentation scheme F and the set of sites S. That is, $A : F \longrightarrow S$. It is a many to many mapping. For each fragment f_i, let ϕ_i be the set of sites it is allocated to. If each of the ϕ_i's is a singleton set, then the allocation scheme is non-replicated. The design for a distributed database is now represented as $(F, A) = \{(f_1, \phi_1), (f_2, \phi_2), \ldots, (f_n, \phi_n)\}$. The allocation scheme can be added to the minimal cover and intersect of the fragment and denote them as $m(f_i^{'}, \phi_i^{'}) = \{(f_j, \phi_j) \mid f_j \bigcap f_i^{'} \neq \emptyset \; where \; f_j \in F\}$ and $int(f_i, \phi_i) = \{(f_j^{'}, \phi_j^{'}) \mid f_i \bigcap f_j^{'} \neq \emptyset \; where \; f_j^{'} \in F^{'}\}$.

5.4 Approaches for Materialization

In this section by employing the representation scheme developed in the previous section a methodology for materialization of a new design for both our approaches the *query generator*

and the *operator method* is developed. A query generator algorithm which generates a set of SQL statements to materialize the new design from the current design is developed. In case of the operator method, algorithms for splitting, reallocating and merging the fragments of the current design to materialize the new design are presented.

The algorithms presented in this section will work for overlapping fragmentation schemes and replicated allocation schemes.

5.4.1 Materialization of a redesigned distributed relational database by automatic generation of SQL commands

As defined in the previous section a grid cell is represented by α_β, where α is the vertical fragment and β is the horizontal fragment the grid cell belongs to. Let $c(\alpha)$ be the set of columns spanning the vertical fragment α, and $p(\beta)$ be the **binding predicate** of the horizontal fragment β. Then the SQL query to define the grid cell is

SELECT $c(\alpha)$ **FROM** R **WHERE** $p(\beta)$. ...(1)

The concatenation of two horizontal grid cells is represented by $(\alpha_\beta, \alpha'_\beta)$, the SQL statement to represent the above well formed expression is

SELECT $c(\alpha) \bigcup c(\alpha')$ **FROM** R **WHERE** $p(\beta)$. ...(2)

Note that for concatenation, forming the union of the column sets of the horizontal grid cells $(\alpha_\beta, \alpha'_\beta)$ is valid.

The union of two vertical grid cells is represented by $(\alpha_{(\beta,\beta')})$. The union of two horizontal fragments β and β' is defined as the disjunction of the binding predicates of the horizontal fragments. The SQL statement which represents this well formed expression is

SELECT $c(\alpha)$ **FROM** R **WHERE** $p(\beta)\, or^2\, p(\beta')$. ...(3)

The fragmentation scheme consists of a set of regular fragments for which, the queries to materialize them need to be generated. Given a regular fragment f' belonging to the fragmentation scheme F', and its representation as $\{\alpha_{1A}, \alpha_{2A}, \ldots, \alpha_{nA}\}$. Then the SQL statement representing the regular fragment is

SELECT $\bigcup_{i=1}^{n} \{c(\alpha_i)\}$ **FROM** R **WHERE** $\bigvee_{a \in A}\{p(a)\}$. ...(4)

[2] Depending upon the predicates the *or* operator can be considered as a part of the SQL syntax. If the two predicates can be combined to form a single predicate, then it is not part of the SQL syntax.

where $\bigvee$ denotes the disjunction operator. The above expression can be derived in a straight-forward manner from the generalization of the basic statements given by **(1), (2) and (3)**.

Let (F, A) be the current design and (F', A') be the new design. Then the algorithm to generate the queries to materialize the new design from the current design is given below

Query_Generator $((F, A), (F', A'))$

1. repeat for each $s \in S$
   (a) define $frg'(s) = \{f_i' \mid s \in \phi_i' \ \& \ (f_i', \phi_i') \in F'\}$
   (b) repeat for each $f' \in frg'(s)$
      i. let representation of f' be $\{\alpha_{1A}, \alpha_{2A}, \ldots, \alpha_{nA}\}$
      ii. Generate SQL statement:
      "**CREATE TABLE** f' *Define_Columns*[3]$(\bigcup_{i=1}^{n}\{c(\alpha_i)\})$;"
      iii. Generate SQL statement:
      "**INSERT INTO** f' $\bigcup_{i=1}^{n}\{c(\alpha_i)\}$
      SELECT $\bigcup_{i=1}^{n}\{c(\alpha_i)\}$
      FROM R
      WHERE $\bigvee_{a \in A}\{p(a)\}$;"
2. repeat for each $s \in S$
   (a) define $frg(s) = \{f_i \mid s \in \phi_i \ \& \ (f_i, \phi_i) \in F\}$
   (b) repeat for each $f \in frg(s)$
      i. Generate SQL statement:
      "**DROP TABLE** f;"

Claim 3 *The algorithm* Query_Generator() *materializes the redesigned distributed database.*

Justification: Each fragment in the new fragmentation scheme F' is materialized at some site. The above algorithm generates the set of SQL statements that must be executed at each of the sites of the distributed database environment. The SQL statements create a fragment (Step 1(b)ii of the above algorithm) of the new fragmentation scheme at its allocated site (Step 1(a) of the algorithm), and insert the data into the fragment using the

[3]Define_Columns() defines the columns according to the syntax of the create DDL statement for the underlying database management system.

"**INSERT INTO TABLE** *table name* **SELECT** ...;" statement (Step 1(b)iii. of the algorithm). The important statement in the above algorithm is the SELECT query in the INSERT statement. The correctness of the above algorithm depends on the correctness of the select statement. The statements **(1), ..., (4)** prove that the above select statement is correct. Hence the above algorithm generates the correct set of statements that need to be executed at each of the sites to materialize the new fragmentation scheme. Note the second repeat loop generates the set of *"Drop table ...;"* statements to drop the current fragmentation scheme. □

The algorithm *Query_Generator* gives us the following set of SQL commands that need to be executed at each of the sites where the fragments of the fragmentation scheme F' are located. The set of SQL commands that need to be executed at site S_3 as given by the *Query_Generator* algorithm are listed below. The representation scheme names for fragments are substituted by actual table names of the local databases by the *Query_Generator* algorithm.

Site S_3

Fragments f_2', f_3' are allocated to site s_3. The set of SQL commands to be executed at site s_3 are:

1. **CREATE TABLE** *EDPS* (
   Emp# Integer,
   Name Char(15),
   City Char(20),
   Dept# Integer,
   Proj# Integer,
   Salary Char(6),
   Bonus Char(6));
2. **CREATE TABLE** *ESB* (
   Emp# Integer,
   Salary Char(6),
   Bonus Char(6));
3. **INSERT INTO** *EDPS*(Emp#, Name, City, Dept#, Proj#, Salary, Bonus)
   SELECT Emp#, Name, City, Dept#, Proj#, Salary, Bonus

FROM Employee

WHERE (20 ≤ Dept# < 30) or (Dept# > 30);

4. **INSERT INTO** *ESB*(Emp#, Salary, Bonus)

   SELECT Emp#, Salary, Bonus

   FROM Employee

   WHERE (Dept# < 10) or (20 ≤ Dept# < 30);

Similarly the *Query_Generator* will specify appropriate SQL statements that need to be executed at Sites S_1, S_2 and S_4 to materialize fragments EDP, $ESD1$ and $ESD2$ respectively. Now a set of commands to drop the fragments belonging to the fragmentation scheme F need to be executed at each of the sites where they are located. The set of commands which are executed at site S_3 are:

1. **Site** S_3

   Drop TABLE *SAL_SCHOOL3*;

Thus execution of the above set of SQL commands at their respective sites will materialize the new fragmentation scheme F'.

5.4.2 Materialization with operations on fragments

Let f be a regular fragment with its representation $r(f) = \{\alpha_{1A}, \alpha_{2A}, ..., \alpha_{nA}\}$, where $\alpha_1, \alpha_2, \ldots, \alpha_n$, are the set of vertical fragments and A consists of the set of horizontal fragments. This representation is denoted as N_A where $N = \bigcup_{i=1}^{n}\{c(\alpha_i)\}$, and $c(\alpha_i)$ is the set of columns spanning the vertical fragment α_i. The operations split and merge are defined on this representation, whereas the operations move, replicate, and delete are independent of the representation. The operation split generates two sub-fragments by splitting the fragment either horizontally or vertically. This gives the following two variations of the split operations:

split_v $(N_A; (N_1)_A, (N_2)_A)$ is the operation which denotes vertically splitting the fragment represented by N_A to generate new sub-fragments represented by N_{1A} and N_{2A} where $N = N_1 \bigcup N_2$.

split_h $(N_A; N_{A_1}, N_{A_2})$ is the operation which denotes horizontally splitting the fragment represented by N_A to generate two sub-fragments represented by N_{A_1} and N_{A_2} where $p(A) = p(A_1) \bigvee p(A_2)$. Note that A_1 and A_2 are horizontal fragments.

Merge is similarly defined as merging horizontally or vertically, vertical merging is equivalent to forming a union of tuples, whereas horizontal merging is equivalent to concatenating the sub-fragments. The merging operations are denoted as follows.

merge_h $(N_{A_1}, N_{A_2}; N_A)$ meaning that the fragments represented by N_{A_1} and N_{A_2} are merged horizontally to form the fragment N_A where $p(A) = p(A_1) \bigvee p(A_2)$.

merge_v $((N_1)_A, (N_2)_A; N_A)$ meaning that the fragments represented by N_{1A} and N_{2A} are merged vertically to form the fragment N_A where $N = N_1 \bigcup N_2$.

For example, fragment f_2 of the fragmentation scheme F shown in Figure 19 has its representation as $(3_{(a,b)}, 4_{(a,b)})$; so given the representation of fragment f' as $4_{(a,b)}$, the operation split_v(f_2; f', f'') splits the fragment f_2 into fragments f' (represented by $4_{(a,b)}$) and f'' (represented by $3_{(a,b)}$). Similarly the operation merge_v(f', f''; f_2), merges fragments f' and f'' to form the fragment f_2. Note that for the split_v operation, the number of tuples in $(N_1)_A$ and $(N_2)_A$ are the same as that of N_A, and vice versa for the merge_v operation. Similarly, the number of columns in N_{A_1} and N_{A_2} are the same as that of N_A for the split_h operation, and vice versa for merge_h operation.

The operations for reallocation are defined as follows.

move $(f, s, s^{'})$ moves the fragment f from site s to site s'.

replicate (f, s) materializes a copy of fragment f at site s.

remove (f, s) removes the fragment f at site s.

Given two designs (F, A) and $(F^{'}, A^{'})$, the methodology to materialize (F', A') from (F, A) can be illustrated as follows:

$$(F, A) \rightarrow \mathit{Divide} \rightarrow (F \bigcap F^{'}, A) \rightarrow \mathit{Relocate} \rightarrow (F \bigcap F^{'}, A^{'}) \rightarrow \mathit{Conquer} \rightarrow (F^{'}, A^{'}).$$

That is, at each of the local sites the fragments of the current fragmentation scheme are split into sub-fragments based on the intersection of the current fragmentation scheme with the new fragmentation scheme (i.e. *Divide*). Next, these sub-fragments are reallocated by comparing the current and new allocation schemes (i.e *Relocate*). Afterwards, the sub-fragments at each of the local sites are merged to form the fragments of the new fragmentation scheme (i.e. *Conquer*).

The $sub_frgs(f)$ consists of the sub-fragments into which the fragment f is split, that is, $sub_frgs(f) = \{f \bigcap f^{'} \mid f^{'} \in int(f)\}$. The number of splits necessary to generate all the sub-fragments of the fragment belonging to fragmentation scheme F is given by $card\{int(f)\}-1$. The split operations necessary to generate all the sub-fragments of the fragments depends on the structure of $sub_frgs(f)$. Note that by Regular Fragment Intersection Closure theorem each of the sub-fragments are regular. Moreover, since the fragmentation schemes are non-overlapping, each of the sub-fragments of the fragment f are disjoint.

In order to materialize a new design, fragments are split to their grid cells and clustered according to their sub-fragment definitions. Then these clusters of sub-fragments are moved according to the new allocation scheme. Note that a fragment consists of a cluster of grid cells which are horizontally and/or vertically merged. This merging sequence is derived from the well formed expression defining the fragment. Using the previously defined notation, the grid cells (α_β) of each of the fragments (f) of the current fragmentation scheme (F) are generated, and then these grid cells (α_β) are clustered according to the elements of the set $sub_frgs(f)$. These clusters are then reallocated according to the new allocation scheme (A'). Once all the grid cells (α_β) for forming the new fragments are allocated according to the new allocation scheme (A'). The fragments (f') are formed by merging the grid cells (α_β) in their representation so as to materialize the new fragmentation scheme (F').

In this case, the following intermediate operations are needed:

break (f) which splits the fragments into its grid cells. This operation is defined in terms of the $split_v$ and $split_h$ operations.

cluster (f) which clusters the set of grid cells of the fragment f into one cluster.

locate ($cluster(f), \phi$) which moves the cluster of grid cells forming the fragment f to set of sites ϕ.

form (f) which forms the fragment f from its grid cells by merging horizontally and/or vertically. Note that this operation will use the operations $merge_h$ and $merge_v$.

Figure 22. illustrates the hierarchy of the materialization operations. The operations $split_h$, $split_v$, $merge_h$ and $merge_v$ are primitive operations. The operations $break$, $relocate$ and $form$ are built on these primitive operations. The efficiency of the operations

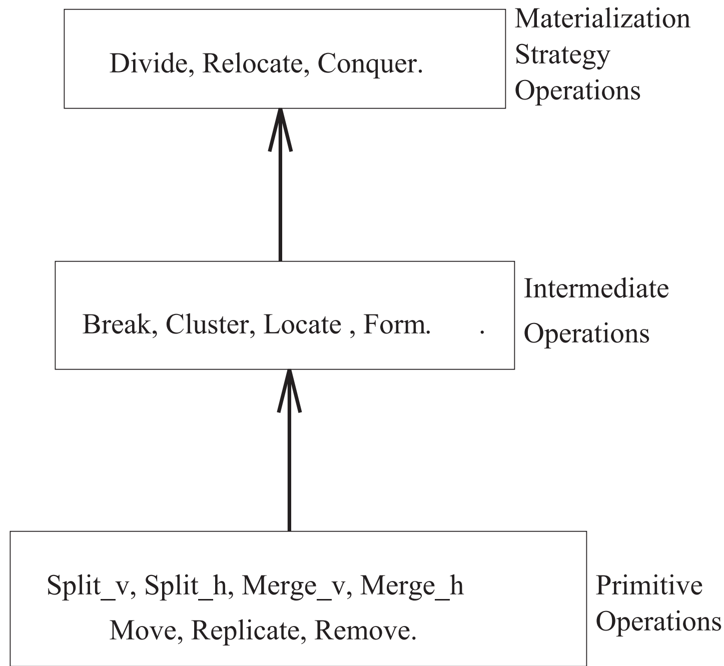


Figure 22: A Hierarchy of Fragment Operations

break and *form* depends on the efficiency of the split and merge operations. The operations $Divide$, *Relocate* and $Conquer$ are the high level operations which are used by the algorithm which materializes the redesign.

The whole process of materialization of the new fragmentation and allocation scheme is done in three phases, namely: *divide, relocate* and *conquer*. The *divide* phase splits the fragments into grid cells and clusters them according to elements of the set $sub_frgs(f)$. The *relocate* phase allocates these clusters according to the new allocation scheme. The *conquer* phase forms the fragments of the new fragmentation based on the clusters of the sub-fragments of the fragments of the new fragmentation scheme. Note that in the *divide* phase the routines $break$ and $cluster$ are used, in the $relocate$ phase the routine *locate* is used, and in the $conquer$ phase the routine $form$ is used. Each of these phases is supported by an algorithm. Hence, the materialization of the new fragmentation and allocation scheme is equivalent to executing the following algorithms.

Divide(F)

1. repeat for each site $s \in S$

   (a) repeat for each $f \in F$ at s

i. break(f)

ii. generate $sub_frgs(f)$

iii. repeat for each $(f \bigcap f') \in sub_frgs(f)$

A. cluster($f \bigcap f'$);

Relocate($F \bigcap F'$)

1. repeat for each site $s \in S$

   (a) repeat for each $f \in F$ at s

   i. repeat for each $(f \bigcap f^{'}) \in sub_frgs(f)$

   A. locate (cluster $(f \bigcap f^{'}), s(f^{'})$[4]);

Conquer($F^{'}$)

1. repeat for each site $s \in S^{'}$

   (a) repeat for each $f^{'} \in F^{'}$ at s

   i. form($f^{'}$);

Claim 4 *The algorithms* Divide(F), Relocate($F \bigcap F'$), and Conquer($F^{'}$) *correctly materialize the redesigned distributed database.*

Justification: The correctness of the materialization by the above three algorithms is supported by the following points:

1. The *Divide* algorithm does the following:

   - All fragments f belonging to fragmentation scheme F at a site are broken down to the set of grid cells by using the function *break*. The *break* function uses the split_v and split_h functions and is described in the Appendix A.
   - These grid cells are then clustered according to the elements of the sets $(f \bigcap f') \in sub_frgs(f)$. Note that the elements of the set $sub_frgs(f)$ are defined by the fragments $f' \in F'$ that have non empty intersection with fragment f. Each cluster $(f \bigcap f')$ is required to form the fragment f' of the new fragmentation

[4] $s(f^{'})$ gives all the sites where the fragment $f^{'}$ is located according to the new allocation scheme $A^{'}$.

scheme F'. Thus the routine *Divide* breaks the fragments $f \in F$, and clusters them to facilitate formation of the fragments $f' \in F'$ at all sites where fragments of fragmentation scheme F are allocated (Step 1(a) of Divide routine).

2. The *Relocate* algorithm does the following:

   - At each site where a fragment $f \in F$ is allocated, for each cluster of grid cells $(f \bigcap f') \in sub_frgs(f)$, if f' is not allocated at this site, then it locates the cluster of grid cells $(f \bigcap f')$ at the site where f' is allocated.
   - At the end of the previous step all the clusters of grid cells are located at sites where the fragments f' that require data from these grid cells are allocated. Moreover, as this routine is executed at all sites, all the grid cells required to form the fragments f' get located at the site where f' is allocated (Step 1(a) of the Relocate routine).

3. The *Conquer* algorithm forms the fragments $f' \in F'$ at each of the sites where they are allocated by merging the grid cells, that define the fragment. Note that the execution of algorithms *Divide, and Relocate* before executing *Conquer* implies that the grid cells are already materialized at the sites where they are needed to form the fragments of the new fragmentation scheme. The routine form uses the operations merge_h and merge_v and is described in Appendix A.

4. Thus the execution of the *Divide, Relocate, and Conquer* algorithms correctly materializes the distributed database. □

The operator method gives the distributed database designer more freedom to efficiently materialize the new fragmentation scheme. This is possible because the above set of operations are supported by the module which is not an integral part of the distributed database system. Once these set of routines have been successfully executed, changes are made in the system directory to represent the new fragmentation scheme. This can be done by executing the SQL statements "CREATE TABLE ..., " as mentioned in the query generator method.

Once the *divide* and *relocate* routines are executed at each of the sites where the fragments of the current design are located, the grid cells required to form the fragments of the

new design (i.e. corresponding to F') will be located at the sites matching the new allocation scheme A'. After this, the conquer algorithm is executed at each of the sites where the fragments of the fragmentation scheme F' are allocated. Figures 18 and 20. show the fragmentation schemes, their representation is shown in Figures 19 and 21. Table 4 shows the *intersect* for each of the fragments of the fragmentation scheme F and Table 3 shows the *minimal cover* for each of the fragments of the fragmentation scheme F'. With reference to the above set of figures the *divide*, *relocate* and *conquer* algorithms will execute the following set of routines. Note that the fragments named as (f_1, f_2, f_3, f_4) for fragmentation scheme F and $(f'_1, f'_2, f'_3, f'_4, f'_5)$ for fragmentation scheme F' will be replaced by the corresponding table names by the materialization of the redesign tool. The routines that need to be executed are listed based on the sites at which the fragments of the fragmentation scheme F are located.

Site S_1

Old design: The fragments $DEPT_PROJ$ and $SAL_SCHOOL2$ are located at site s_1.
New design: EDP is allocated to site s_1.
$(int(DEPT_PROJ)) = \{EDP, EDPS\}$.
$(sub_frgs(DEPT_PROJ)) = \{DEPT_PROJ \bigcap EDP, DEPT_PROJ \bigcap EDPS\}$.
$(int(SAL_SCHOOL2)) = \{EDPS, ESD1\}$.
$(sub_frgs(SAL_SCHOOL2)) = \{SAL_SCHOOL2 \bigcap EDPS, SAL_SCHOOL2 \bigcap ESD1\}$.
However, $DEPT_PROJ \bigcap EDP \neq \emptyset$. Hence $cluster(DEPT_PROJ \bigcap EDP)$ and $locate(cluster(DEPT_PROJ \bigcap EDP), s_1)$ are not listed in the routines which are run.

1. break($DEPT_PROJ$);
2. break($SAL_SCHOOL2$);
3. cluster($DEPT_PROJ \bigcap EDPS$);
4. locate(cluster($DEPT_PROJ \bigcap EDPS$), s_3);
5. cluster($SAL_SCHOOL2 \bigcap EDPS$);
6. cluster($SAL_SCHOOL2 \bigcap ESD1$);
7. locate(cluster($SAL_SCHOOL2 \bigcap EDPS$), s_3);

8. locate(cluster($SAL_SCHOOL2 \bigcap ESD1$), s_2);

Site S_2

Old design: The fragment $EMP_SCHOOL1$ is located at site s_2.
New design: $ESD1$ is allocated to site s_2.
$(int(EMP_SCHOOL1)) = \{ESB, ESD1\}$.
$(sub_frgs(EMP_SCHOOL1)) = \{SAL_SCHOOL2 \bigcap ESB, SAL_SCHOOL2 \bigcap ESD1\}$.
However, $EMP_SCHOOL1 \bigcap ESD1 \neq \emptyset$. Hence $cluster(EMP_SCHOOL1 \bigcap ESD1)$ and $locate(cluster(EMP_SCHOOL1 \bigcap ESD1), s_2)$ are not listed in the routines which are run.

1. break($EMP_SCHOOL1$);
2. cluster($EMP_SCHOOL1 \bigcap ESB$);
3. locate(cluster($EMP_SCHOOL1 \bigcap ESB$), s_3);

Site S_3

Old design: The fragment $SAL_SCHOOL3$ is located at site s_3.
New design: $EDPS$ is allocated to site s_3.
$(int(SAL_SCHOOL3)) = \{EDPS, ESD2\}$.
$(sub_frgs(SAL_SCHOOL3)) = \{SAL_SCHOOL3 \bigcap EDPS, SAL_SCHOOL3 \bigcap ESD2\}$.
However, $SAL_SCHOOL3 \bigcap EDPS \neq \emptyset$. Hence $cluster(SAL_SCHOOL3 \bigcap EDPS)$ and $locate(cluster(SAL_SCHOOL3 \bigcap EDPS), s_3)$ are not listed in the routines which are run.

1. break($SAL_SCHOOL3$);
2. cluster($SAL_SCHOOL3 \bigcap ESD2$);
3. locate(cluster($SAL_SCHOOL3 \bigcap ESD2$), s_5);

Now the set of *form* routines to be executed at sites as specified by the *conquer* algorithm are listed.

Site S_1

New design: Fragment EDP is located at site s_1. Hence the following routine is executed at this site.

1. form(EDP);

Site S_2

New fragment $ESD1$ is located at site s_2. Hence the following routine is executed at this site.

1. form($ESD1$);

Site S_3

New fragment $EDPS$ and ESB are located at site s_2. Hence the following routine is executed at this site.

1. form($EDPS$);
2. form(ESB);

Site S_4

New fragment $ESD2$ is located at site s_4. Hence the following routine is executed at this site.

1. form($ESD2$);

Once these set of routines are executed, the new fragmentation scheme will be materialized. Though the routines to be run are listed according to the sites where the fragments are located, it should be possible to run these set of routines by means of a remote procedure call. Now changes need to be made in the system directory to provide users and applications access to the new design. The above example illustrates how the two methods materialize the new design on the basis of the current design. This transfer from an existing distribution of data to a new distribution of data can be done automatically.

5.5 Extension

In the last section algorithms that materialize the redesigned distributed databases using either SQL statements or using operators on fragments had been developed. But the change in the grid structure has not been taken into consideration. That is, now the redesign process

using the mixed fragmentation methodology generates a set of grid cells based on different vertical and horizontal fragmentation schemes.

Let (F, A) be the current design based on vertical fragments $1, 2, \ldots, n$ and horizontal fragments $a, b, \ldots, x$ and denoted as $\{(f_1, \phi_1), (f_2, \phi_2), \ldots, (f_p, \phi_p)\}$.

Let $(F^{'}, A^{'})$ be the new design based on vertical fragments $V^{'} = (1^{'}, 2^{'}, \ldots, m^{'})$ and horizontal fragments $H^{'} = (a^{'}, b^{'}, \ldots, y^{'})$ and denoted as $\{(f_1^{'}, \phi_1^{'}), (f_2^{'}, \phi_2^{'}), \ldots, (f_q^{'}, \phi_q^{'})\}$.

Define a set of virtual vertical fragments $V'' = \{c(v'') = c(k) \cap c(k^{'}) \mid c(k) \cap c(k^{'}) \neq \emptyset$, where $k \in V$ and $k^{'} \in V^{'}\}$, and a set of virtual horizontal fragments $H'' = \{p(h'') = p(u) \bigwedge p(u^{'}) \mid p(u) \bigwedge p(u^{'})$ is non-contradictory[5] , where $u \in H$ and $u^{'} \in H^{'}\}$

Let G be the set of grid cells formed by vertical fragmentation scheme (V) and horizontal fragmentation scheme (H), $G^{'}$ be the set of grid cells formed by vertical fragmentation scheme $(V^{'})$ and horizontal fragmentation scheme $(H^{'})$ and G'' be the set of virtual grid cells generated by the virtual vertical fragmentation scheme (V'') and virtual horizontal fragmentation scheme (H''). The grid cells G are mapped to grid cells G'' and the grid cells G' are mapped to grid cells G'', so that the fragmentation schemes F and $F^{'}$ are now defined based on the common set of virtual grid cells G''. Then the algorithms presented in the last section can be applied to materialize new design $(F^{'}, A^{'})$ from the current design (F, A) as both the fragmentation schemes are defined on the same set of virtual grid cells.

Theorem 5 Mapping of grid cells: *Any grid cell $\alpha_\beta \in G$ can be mapped to a mixed fragment f'' defined on grid cells G''.*

Proof: Consider the grid cell α_β, i.e., α is the vertical fragment in $\{1, 2, \ldots, n\}$, and β is the horizontal fragment in $\{a, b, \ldots, x\}$.

The grid cell α_β shall be mapped to the virtual grid cells defined by G''.

Let $\Gamma = \{v'' \mid c(\alpha) \cap c(v'') \neq \emptyset$ where $v'' \in V'' \}$.

Let $\Delta = \{h'' \mid p(\beta) \cap p(h'')$ is non-contradictory where $h'' \in H'' \}$.

Because of the way in which V'' and H'' are defined, $c(\alpha) \cap c(v'') = c(v'')$ for all $v'' \in \Gamma$ and $p(\beta) \bigwedge p(h'') \Rightarrow p(h'')$ for all $h'' \in \Delta$. This implies that the set of vertical fragments Γ and horizontal fragments Δ generate complete and non-overlapping vertical and horizontal fragments of α and β respectively.

[5] Non-contradictory means that conjunction of the two predicates $p(u)$ and $p(u^{'})$ makes sense, unlike conjunction of predicates, say, (SAL $<$ 20k) and (SAL $>$ 40K) which is contradictory because it does not make sense as there will be no tuple that satisfies both the predicates.

Moreover, $\bigcup_{v'' \in \Gamma} c(v'') = \alpha$, and $\bigvee_{h'' \in \Delta} p(h'') = \beta$.

Thus, the set of grid cells formed by vertical fragments in Γ and horizontal fragments in Δ that completely define grid cell α_β belonging to G. The mixed fragment formed by the grid cells generated by Γ and Δ represents the grid cell α_β belonging to G.

Hence any grid cell in G can be represented by a mixed fragment f'' formed by grid cells in G'' whose representation is $r(f'') = \{v''_{h''} \mid v'' \in \Gamma, h'' \in \Delta\}$. □

By the above theorem, there is a mapping from each of the grid cells in G and G' to mixed fragments defined on grid cells in G''. Similarly, the fragments in fragmentation scheme F can be mapped to mixed fragments on grid cells in G'' and fragments in fragmentation scheme F' to mixed fragments on grid cells in G''.

Therefore, the fragments in fragmentation schemes F and F' are defined on a common set of grid cells G''; hence the materialization algorithms developed in last section can be used for materializing the redesigned distributed databases.

Taking the change of grid cells into consideration will incur additional cost as the virtual vertical and horizontal fragments need to be generated, thus creating a common representation to the fragmentation schemes, before the materialization algorithms developed in Section 3 can be applied.

5.6 Cost Model

In this section a cost model for materializing a redesigned distributed database is developed. This cost model will be based on the amount of time taken to materialize the new redesigned distributed database. Since the size of the distributed database is going to be large, only the time spent on data access and communication delay incurred to materialize the new design will be considered while developing the cost model. Both these costs are based on the size of the fragments of the distribution design.

The fragmentation scheme is based on the mixed fragmentation approach. Therefore, an algorithm to calculate the size of the mixed fragments given the sizes of the vertical and horizontal fragments can be presented.

Let $\{1, 2, 3, \ldots, n\}$ be the set of vertical fragments of a relation R. Let K be the key of the relation. Let $l(1), l(2), \ldots, l(n)$ be the lengths of the vertical fragments *excluding the length of the key* and $l(K)$ be the length of the key. Let $\{a, b, \ldots, x\}$ be the set of horizontal

fragments of the relation. Let $\{t(a), t(b), \ldots, t(x)\}$ be the number of tuples in each of the horizontal fragments respectively.

Then the size of a regular mixed fragment can be calculated iteratively as follows:

1. The size of the grid cell represented by α_β is $v(\alpha_\beta) = \{l(\alpha) + l(K)\} \times t(\beta)$.
2. The size of the fragment represented by $\alpha_\beta \cup \alpha_{\beta'}$ is $v(\alpha_\beta \cup \alpha_{\beta'}) = \{l(\alpha) + l(K)\} \times \{t(\beta) + t(\beta'))\}$.
3. The size of the fragment represented by $\alpha_\beta \| \alpha'_\beta$ is $v(\alpha_\beta \| \alpha'_\beta) = \{l(\alpha) + l(\alpha') + l(K)\} \times t(\beta)$.
4. The size of a regular mixed fragment f with its representation $r(f)$ as $(\alpha_{1A}, \alpha_{2A}, \ldots, \alpha_{pA})$ is $v(f) = \{\sum_{i=1}^{i=p}(l(\alpha_i) + l(K))\} \times \{\sum_{\beta \in A}(t(\beta))\}$.

The above set of iterative rules provide us with an algorithm to calculate the size of each of the fragments of fragmentation schemes F and F'. Similarly, the sizes of the sub-fragments of fragment based on intersect and cover operations can be calculated. The sizes of fragments and its sub-fragments enable us to calculate the time taken to materialize a new redesigned distributed database.

Given (F, A) and (F', A') as the current and new redesigned distribution designs. Let $f \in F$ be a fragment that is needed by fragments $f' \in int(f)$. This requires fragment f to be split into $card(int(f))$ sub-fragments. The cost of doing this is:

$$\{\lceil \frac{v(f)}{pagesize} \rceil + \sum_{f' \in int(f)} \lceil \frac{v(f \cap f')}{pagesize} \rceil\} \times \{DataAccessTime\} \tag{2}$$

where *pagesize* is the size of the page in bytes holding the tuples of the relation, *DataAccessTime* is the time taken on average to retrieve page of data from the stable storage to the main memory. The above cost formula gives the time taken to *divide* a fragment $f \in F$ to its sub-fragments based on fragmentation scheme F'. Each page that is read into the main memory is split into the vertical fragments defined by intersecting sub-fragments and placed in the pages corresponding to horizontal fragments of the intersecting sub-fragments $int(f)$.

A fragment $f' \in F'$ that needs data from fragments $f \in m(f')$ (i.e. minimal cover), requires f' to be merged from $card(m(f'))$ sub-fragments. The cost of doing this is:

$$\{\lceil\frac{v(f')}{pagesize}\rceil + \sum_{f\in m(f')} \lceil\frac{v(f\cap f')}{pagesize}\rceil\} \times DataAccessTime \tag{3}$$

The above formula gives us the cost of merging the sub-fragments of the new fragment f' based on the initial fragmentation scheme F. It is assumed that the sub-fragments are stored as temporary tables in the site where f' is to be located. This a valid assumption because these sub-fragments are mostly transferred from other sites of the distributed database environment.

The cost of moving the sub-fragments according to the new allocation scheme is given by:

$$\sum_{f\in m(f')\&s(f)\neq s(f')} \frac{v(f\cap f')}{DataTransferRate}, \tag{4}$$

where the *DataTransferRate* is the number of bytes that can be transferred on an average from one site to another in the distributed database environment. Only those sub-fragments of a fragment f needed at other sites are transferred.

Finally, the cost of materializing the redesigned distributed database defined by (F', A') based on the current design (F, A) by using the operator method is given by:

$$\begin{aligned} C_o \;=\; & \{\sum_{f\in F}\{\lceil\frac{v(f_i)}{pagesize}\rceil + \sum_{f'\in int(f)} \lceil\frac{v(f_i\cap f')}{pagesize}\rceil\} \\ & + \sum_{f'\in F'}\{\lceil\frac{v(f'_j)}{pagesize}\rceil + \sum_{f\in m(f')} \lceil\frac{v(f\cap f'_j)}{pagesize}\rceil\}\} \times \{DataAccessTime\} \\ & + \sum_{f'\in F'\&f\in m(f')\&s(f)\neq s(f')} \{\frac{v(f\cap f')}{DataTransferRate}\} \end{aligned} \tag{5}$$

The first part of the equation gives the cost of the **Divide** phase, the second part gives the cost of the **Conquer** phase and the last part gives the cost of the **Relocate** phase. These cost functions have been generalized from the equations 2, 3 and 4. Given the length of the tuple of each vertical fragment, the cardinality of each horizontal fragment, the data access time and the data transfer rate, equation 5 gives the cost of materializing the design (F', A') using the operator method.

In case of the query generator approach, a set of SQL queries are executed at each of the sites of the distributed database environment. The set of SQL statements basically

perform the operations of *Divide* and *Conquer*. Although, it is not obvious from evaluating the SQL statements at each site, when all the SQL statements generated by query generator accessing a fragment of a given fragmentation scheme are considered together it becomes fairly obvious that the set of SQL statements perform both *Divide* and *Conquer* operations.

The distributed query processor of the distributed database system does not have the capability of processing a set of SQL statements. Hence it evaluates and processes each of the SQL statements to its completion. Thus a set of SQL statements that divide a fragment f into its sub-fragments $int(f)$ would require a complete scan of the fragment f for each of the sub-fragments generated. But when merging the sub-fragments to form a new fragment, the sub-fragments (which already exist either as temporary tables or relations) are loaded into the main memory and the operations union or join are performed. Thus the cost of merging the sub-fragments to form a new fragment is exactly the same as that for the operator method. The sub-fragments of a fragment f are are located at sites where they are needed to form the new fragments. Again the cost of doing this will be the same as that for the operator method. Therefore the cost of materializing the redesigned distributed database by query generator method is given by:

$$\begin{aligned} C_q &= \{\sum_{f\in F}\sum_{f'\in int(f)}\{\lceil\frac{v(f_i)}{pagesize}\rceil + \lceil\frac{v(f_i\cap f')}{pagesize}\rceil\} \\ &+ \sum_{f'\in F'}\{\lceil\frac{v(f_j')}{pagesize}\rceil + \sum_{f\in m(f')}\lceil\frac{v(f\cap f_j')}{pagesize}\rceil\}\} \times \{DataAccessTime\} \\ &+ \sum_{f'\in F' \& f\in m(f') \& s(f)\neq s(f')}\{\frac{v(f\cap f')}{DataTransferRate}\} \end{aligned} \quad (6)$$

Note that the query generator method incurs the cost of scanning the entire fragment to generate each of its sub-fragments. Here it is assumed that there are no indexes that can be used to reduce the number of pages that need to be accessed. Even if for one sub-fragment generation an index cannot be used, the cost of materialization for the query generator method will be larger than the operator method. But in the case of the operator method the complete fragment is scanned only once to generate all its sub-fragments, whereas the query generator method may require multiple scans of a fragment to generate all its sub-fragments. The only case when the costs are the same is when there is only limited redesign. It is also not practical to have indexes to benefit the generation of the sub-fragments because of lack of disk space, and also index generation is a costly and time consuming job.

Therefore, the operator method will perform better than query generator method almost always.

5.7 Summary

Two **automated** approaches (*Query Generator method* and *Operator method*) to materialize the new designs were described. The algorithms were presented and their correctness proved for both the methods. By means of an example we illustrated how the materialization procedure is automatically done by both approaches. The query generator method depends on the distributed database system's processing capabilities for efficiently materializing the new designs. Thus the efficiency of materialization in this approach depends on the global query optimization, decomposition and execution. In case of small sized distributed databases (possibly up to 10's of gigabytes in size) in a LAN environment, the query generator method will be useful. But, in case of large databases with hundreds of gigabytes of data, the distributed database system may not be able to handle the volume of data transferred.

On the other hand, the operator method is independent of the distributed database system functionality. The operator method is more suitable for large distributed databases in a WAN environment. The major advantage of the operator method is that it can initiate the routines *divide*, *relocate* and *conquer* in background with minimum interaction from the distributed database system. Moreover, the operations *split*, *merge* and *move* can be implemented so as to be efficient, it is this control which is lacking in the query generator method (which is very system dependent). It is this flexibility in implementation of the primitive operations that makes the operator method more general and powerful than the query generator method.

The materialization methodology was extended to take into consideration the change in grid structure during the redesign process. This extension is done by defining both the current and new design by means of a common set of virtual grid cells. This enables us to use the materialization algorithms that were developed for the case when there is no change in grid structure. A cost model based on the sizes of the horizontal and vertical grid fragments is developed to estimate the time taken to materialize the distributed database by using both methods. It was shown that the query generator approach is in general more costly than the operator approach.

CHAPTER 6

EFFICIENT MATERIALIZATION BY USING THE QUERY GENERATOR APPROACH

6.1 General Comments

In the last chapter two methods to materialize the redesigned distributed databases were presented. The cost model was developed, which showed that the operator approach is more efficient than query generator approach, if for accessing at least one fragment no index could be used. In this chapter a multiple query optimization technique will be used to enhance the efficiency of the query generator approach to materialize the redesigned distributed databases.

6.2 Multiple Query Optimization

The query generator approach to materialize the redesigned distributed database initiates a set of SQL statements at various sites of the distributed database environment. Since all these SQL statements are going to executed simultaneously, there is a scope for optimizing them. Moreover, the fragments and grid cells that need to be accessed by these SQL statements are known in advance. The SQL statements to materialize the fragments of the new distributed database design are all sent to a coordinator site. This coordinator site uses the multiple query optimization technique (as described below), to generate a set of SQL statements each of which need to be executed at various sites of the distributed database environment. This process of enhancing the efficiency of the query generator approach using the multiple query optimization is described by means of the following steps:

1. Each SQL statement is rewritten to specify which fragments of the current distributed database design are accessed.
2. After this each SQL statement is parsed and a query graph is generated.

3. This query graph is modified and extended by adding *get* and *combine* operations. The leaf nodes of the query graph are the fragments of the current distributed database design. The root nodes are the fragments of the new distributed database design.

4. These enhanced query graphs are merged into one query graph by merging the common leaf nodes. This merged query graph contains as its leaf nodes all the fragments of current distributed database design, and as the root nodes the fragments of the new distributed database design.

5. Finally, an SQL statement is generated for each leaf node and each root node. There are some commands for transferring the data from one site to another according to the new allocation scheme.

An example will be presented to illustrate these operations. After this a set of algorithms will be described for this approach.

6.3 Example Illustrating use of MQO

Consider the Employee relation given in Figure 16. There are two distributed database designs generated. Figures 18 and 20 illustrate the current design (F, A) and (F', A') respectively. The objective is to use the MQO technique in materializing design (F', A') from the design (F, A) using the query generator algorithm.

Following are the set of SQL statements initiated to materialize the new design:

At site s_1:

INSERT INTO EDP Emp#, Name, City, Dept#, Proj#
SELECT Emp#, Name, City, Dept#, Proj#
FROM Employee
WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20;

At site s_3:

INSERT INTO EDPS Emp#, Name, City, Proj#, Dept#, Salary, Bonus
SELECT Emp#, Name, City, Proj#, Dept#, Salary, Bonus
FROM Employee
WHERE Dept# $\geq$ 30 AND 20 $\leq$ Dept# $<$ 30;

At site s_3:

INSERT INTO ESB Emp#, Salary, Bonus
SELECT Emp#, Salary, Bonus
FROM Employee
WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20;

At site s_2:

INSERT INTO ESD1 Emp#, School, Degree
SELECT Emp#, School, Degree
FROM Employee
WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20 AND 20 $\leq$ Dept# $<$ 30;

At site s_4:

INSERT INTO ESD2 Emp#, School, Degree
SELECT Emp#, School, Degree
FROM Employee
WHERE Dept# $\geq$ 30;

The above set of SQL statements can be rewritten by taking into consideration the sites at which the new fragments need to be materialized and the minimal covers for each of the new fragments given in Table 3. Note that the relation being specified is no longer Employee for all the SELECT statements, but the relations in the respective minimal covers. For example, for materializing the relation EDP in the new design, the relation to be accessed is DEPT_PROJ.

---

1). insert@s_1 EDP
SELECT Emp#, Name, City, Dept#, Proj#
FROM DEPT_PROJ@s_1
WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20;

2). insert@s_3 EDPS
SELECT Emp#, Name, City, Proj#, Dept#, Salary, Bonus
FROM DEPT_PROJ@s_1, SAL_SCHOOL1@s_2, SAL_SCHOOL3@s_3
WHERE Dept# $\geq$ 30 AND 20 $\leq$ Dept# $<$ 30;

3). insert@s_3 ESB

SELECT Emp#, Salary, Bonus

FROM SAL_SCHOOL2@s_1

WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20;

4). insert@s_2 ESD1

SELECT Emp#, School, Degree

FROM SAL_SCHOOL1@s_2,SAL_SCHOOL2@s_1

WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20 AND 20 $\leq$ Dept# $<$ 30;

5). insert@s_4 ESD2

SELECT Emp#, School, Degree

FROM SAL_SCHOOL3@s_3

WHERE Dept# $\geq$ 30;

Note that "insert@s_1 EDP" describes the operation of inserting tuples into the relation EDP at site s_1. Instead of referring to the EMPLOYEE relation in the FROM clause, the set of relations in the minimal cover of the fragments in the new scheme are referred. This immediately restricts the access to those relations which have at least one tuple to form the new fragments.

In the next optimization step the columns and predicates specified in the SELECT statements are replaced by the representation of the corresponding vertical and horizontal fragments. The FROM clause of an SQL statement is replaced by the *combine* function, the SELECT and WHERE clauses are replaced by the *get* function. The function *combine* merges a set of grid cells to form a mixed fragment. The function *get* retrieves a set of grid cells from a mixed fragment. In fact, the functions *combine* and *get* are used to describe the processing done to materialize a fragment by using an intermediate representation. Later the specification of the materialization process in this representation is translated back to a set of SQL statements.

---

1). insert@s_1 EDP

combine($(1_{(a,b)}, 2_{(a,b)})$)

get($(1_{(a,b)}, 2_{(a,b)})$)

from DEPT_PROJ@s_1

2). insert@s_3 EDPS

combine($(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$)

get($(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$)

from DEPT_PROJ@s_1, SAL_SCHOOL2@s_1, SAL_SCHOOL3@s_3

3). insert@s_3 ESB

combine($(3_{(a,b)})$)

get($(3_{(a,b)}$)

from SAL_SCHOOL2@s_1

4). insert@s_2 ESD1

combine($(4_{(a,b,c)})$)

get($(4_{(a,b,c)})$)

from SAL_SCHOOL1@s_2,SAL_SCHOOL2@s_1

5). insert@s_4 ESD2

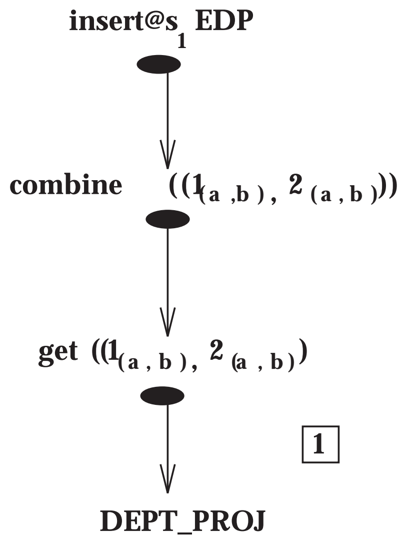
combine($(4_{(d)})$)

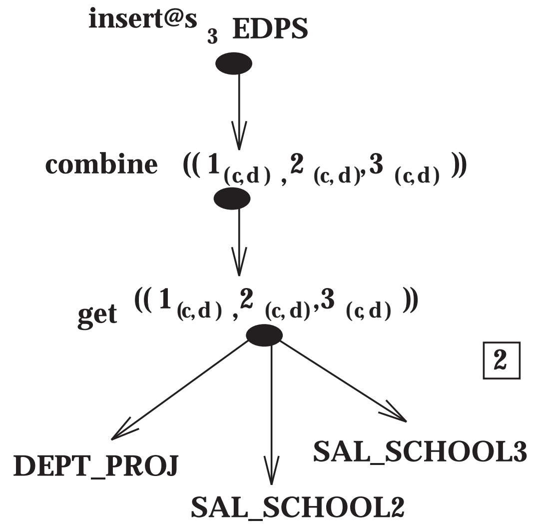
get($(4_{(d)})$)

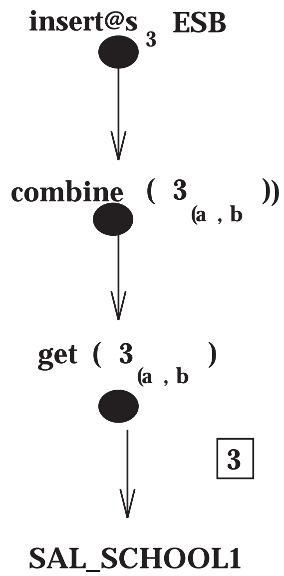
from SAL_SCHOOL3@s_3

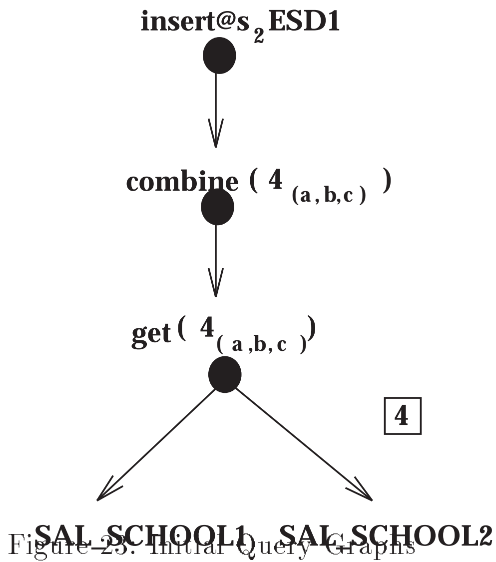
The above set of modified SQL statements describe the functions that need to be performed to materialize the fragments of the new distributed database design. The representation of the fragments are used as the parameters of the functions *get* and *combine*. The function *get* retrieves a set of grid cells from the relation. The function *combine* merges a set of grid cells to form a new fragment. Figure 23 shows the 5 modified query graphs corresponding to the above 5 modified query statements. Note that all the fragments being retrieved are regular fragments.

In the above representation of the SQL queries generated, a new fragment (for example EDPS) is formed by combining data from more than one relation. In order to get the relevant data from each of these relations, the get operation must be propagated to these relations. The query graphs after this modification are shown in Figure 24. Only those query graphs which access data from more than one relation are showed. For the query graphs that access data from only one relation there is no change.









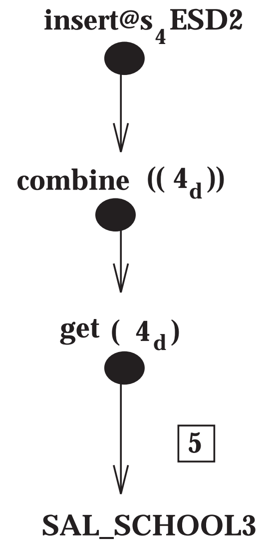
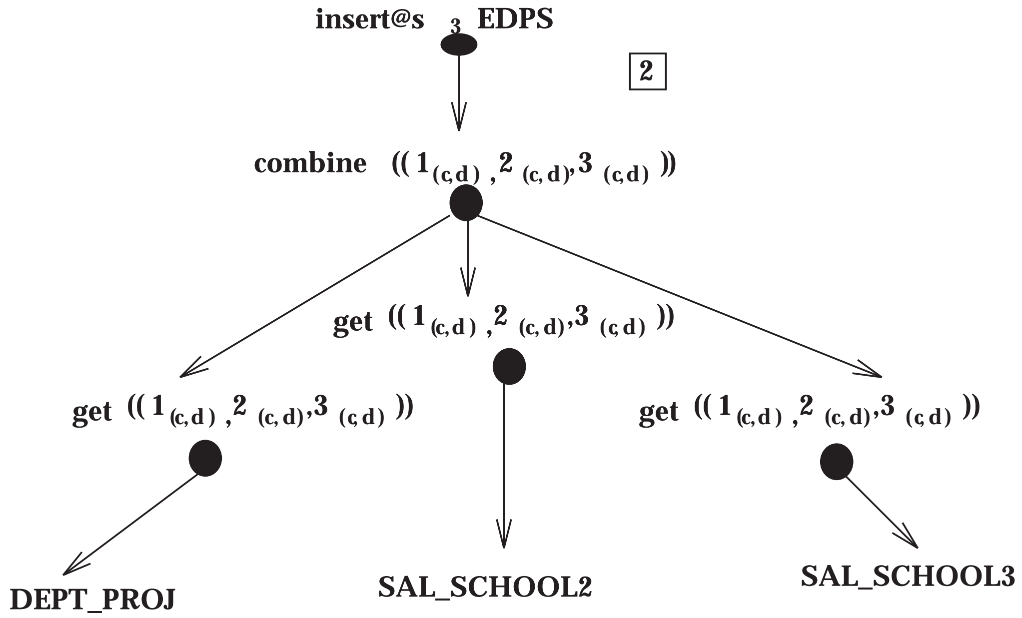


Figure 23: Initial Query Graphs



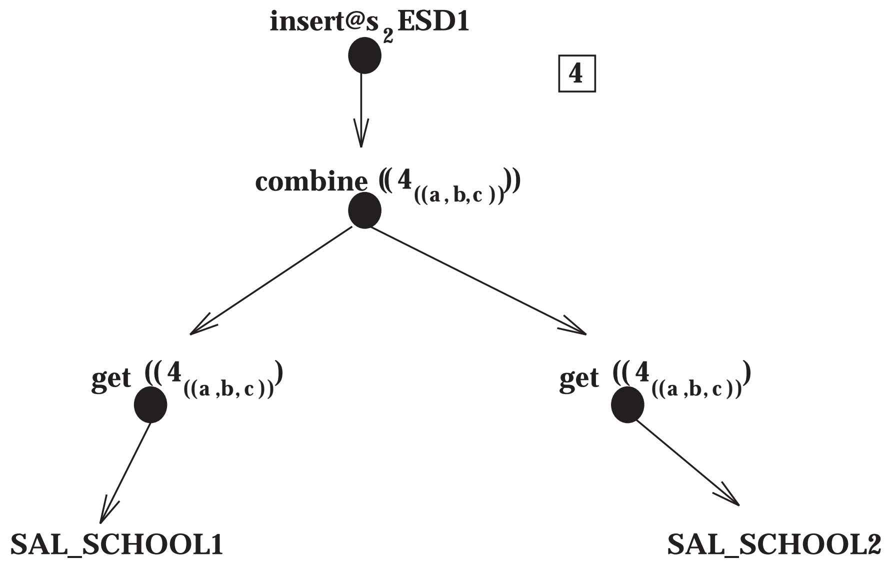


Figure 24: Step 1 of Multiple Query Optimization

1). insert@s_1 EDP

combine($(1_{(a,b)}, 2_{(a,b)})$)

get($(1_{(a,b)}, 2_{(a,b)})$)

from DEPT_PROJ@s_1

2). insert@s_3 EDPS

combine($(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$)

(i) get($(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$)

from DEPT_PROJ@s_1

(ii) get($(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$)

from SAL_SCHOOL2@s_1

(iii) get($(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$)

from SAL_SCHOOL3@s_3

3). insert@s_3 ESB

combine($(3_{(a,b)})$)

get($(3_{(a,b)}$)

from SAL_SCHOOL2@s_2

4). insert@s_2 ESD1

combine($(4_{(a,b,c)})$)

(i) get($(4_{(a,b,c)})$)

from SAL_SCHOOL1@s_2

(ii) get($(4_{(a,b,c)})$)

from SAL_SCHOOL2@s_1

5). insert@s_4 ESD2

combine($(4_{(d)})$)

get($(4_{(d)})$)

from SAL_SCHOOL3@s_3

The above set of modified SQL statements will be optimized further by specifying only the relevant data that needs to be extracted from each of the component relations. This is the next step of query modification, where the sub fragments given by the intersection of the fragment in the *get* expression and representation of the relation accessed is the actual fragment that is retrieved from that relation. For example, the set

of grid cells to be retrieved from DEPT_PROJ is $(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$, and the representation of DEPT_PROJ is $(1_{(a,b,c,d)}, 2_{(a,b,c,d)})$, hence the actual set of grid cells retrieved is $(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)}) \bigcap (1_{(a,b,c,d)}, 2_{(a,b,c,d)})$, which is $(1_{(c,d)}, 2_{(c,d)})$. This optimization step is illustrated in the Figure 25 and the modified query descriptions follow.

---

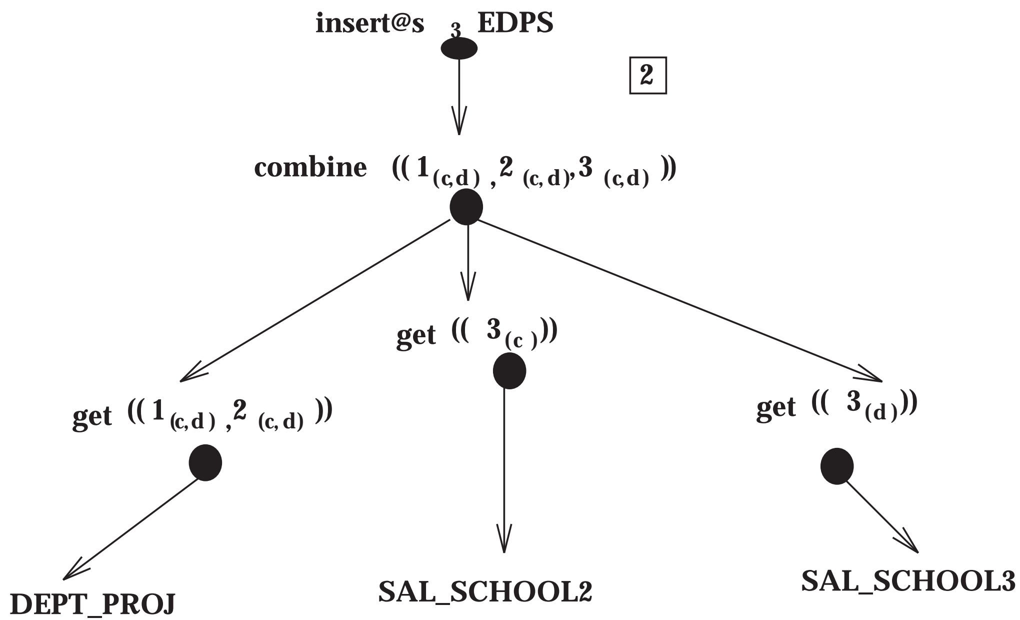
1). insert@s_1 EDP
 combine($(1_{(a,b)}, 2_{(a,b)})$)
 get($(1_{(a,b)}, 2_{(a,b)})$)
 from DEPT_PROJ@s_1

2). insert@s_3 EDPS
 combine($(1_{(c,d)}, 2_{(c,d)}, 3_{(c,d)})$)
 (i) get($(1_{(c,d)}, 2_{(c,d)})$)
  from DEPT_PROJ@s_1
 (ii)get((3_c))
  from SAL_SCHOOL2@s_1
 (iii)get((3_d))
  from SAL_SCHOOL3@s_3

3). insert@s_3 ESB
 combine($(3_{(a,b)})$)
 get($(3_{(a,b)}$)
 from SAL_SCHOOL1@s_2

4). insert@s_2 ESD1
 combine($(4_{(a,b,c)})$)
 (i) get($(4_{(a,b)})$)
  from SAL_SCHOOL1@s_2
 (ii)get((4_c))
  from SAL_SCHOOL2@s_1

5). insert@s_4 ESD2
 combine($(4_{(d)})$)
 get($(4_{(d)})$)
 from SAL_SCHOOL3@s_3



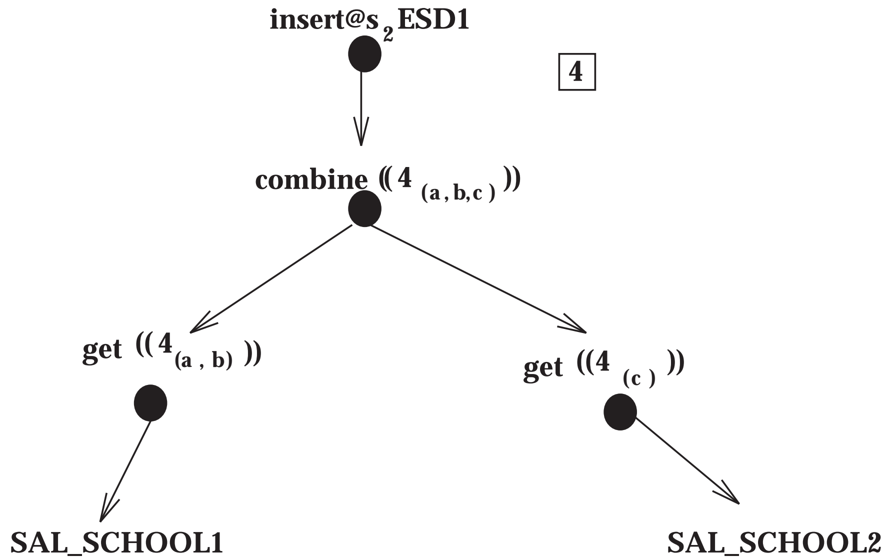


Figure 25: Step 2 of Multiple Query Optimization

The materialization of the new distributed database design is done in two phases, the first phase consists of extracting the relevant data from the component relations and sending it to the sites where it is required. The operation *send_to*(*site_id*), sends a file to the site *site_id*. The second phase consists of combining the data to form the new fragments and inserting in the corresponding relations. The first phase which retrieves the relevant grid cells from the fragments in the minimal cover of the current design and stores them in the temporary tables is described below.

---

1). get($(1_{(a,b)}, 2_{(a,b)})$) $\wedge$ send_to(s_1) as $TempTable_1$,
   get($(1_{(c,d)}, 2_{(c,d)})$) $\wedge$ send_to(s_3) as $TempTable_2$
   from DEPT_PROJ@s_1

2). get($(4_{(a,b)})$) $\wedge$ send_to(s_2) as $TempTable_3$,
   get($(3_{(a,b)})$) $\wedge$ send_to(s_3) as $TempTable_4$
   from SAL_SCHOOL1@s_2

3). get((3_c)) $\wedge$ send_to(s_3) as $TempTable_5$,
   get((4_c)) $\wedge$ send_to(s_2) as $TempTable_6$
   from SAL_SCHOOL2@s_1

4). get((3_d)) $\wedge$ send_to(s_3) as $TempTable_7$,
   get($(4_{(d)})$) $\wedge$ send_to(s_4) as $TempTable_8$
   from SAL_SCHOOL3@s_3

The set of statements below specify the actions that need to be taken at the sites of the fragments of the new distributed database design. Note that the operation *send_to* sends the temporary tables to the sites where the respective new fragments need to be materialized. These statements are self explanatory.

1'). insert@s_1 EDP
   combine($TempTable_1$)

2'). insert@s_3 EDPS

combine($TempTable_2$, $TempTable_5$, $TempTable_7$)

3′). insert@s_3 ESB

combine($TempTable_4$)

4′). insert@s_2 ESD1

combine($TempTable_3$,$TempTable_6$)

5′). insert@s_4 ESD2

combine($TempTable_8$)

The above set of operations are generated by the optimization process for the initial set of queries generated to materialize the redesigned distributed databases. These operations are again translated to a set of SQL statements that need to be executed at each of the sites of the distributed database environment. The set of SQL statements are enclosed between constructs begin{concurrent} and end{concurrent}, and are executed simultaneously. Figure 26 shows the combined query graph. The central nodes of the graph are the *get* nodes, these nodes retrieve the relevant set of grid cells from a fragments of current design to be used in materializing a fragment of the new design. Hence there are arrows coming in from the fragments of the current design showing which grid cells are being accessed, and there are arrows going into the combine statements which specify the grid cells being used to form the fragments of the new design.

The modified query expressions specified above are translated into SQL statements to define the temporary tables that are generated, and to merge (i.e., use the relational algebra operations union and join to combine the temporary tables to form a mixed fragment) these temporary tables to form the fragments of the new design. Thus this optimization procedure generates a set of SQL statements that need to be executed at the sites of the fragments of the current design, and at the sites of the fragments of the new design.

At site s_1:

begin{concurrent}

INSERT INTO $TempTable_1$ Emp#, Name, City, Dept#, Proj#

SELECT Emp#, Name, City, Dept#, Proj#

FROM DEPT_PROJ

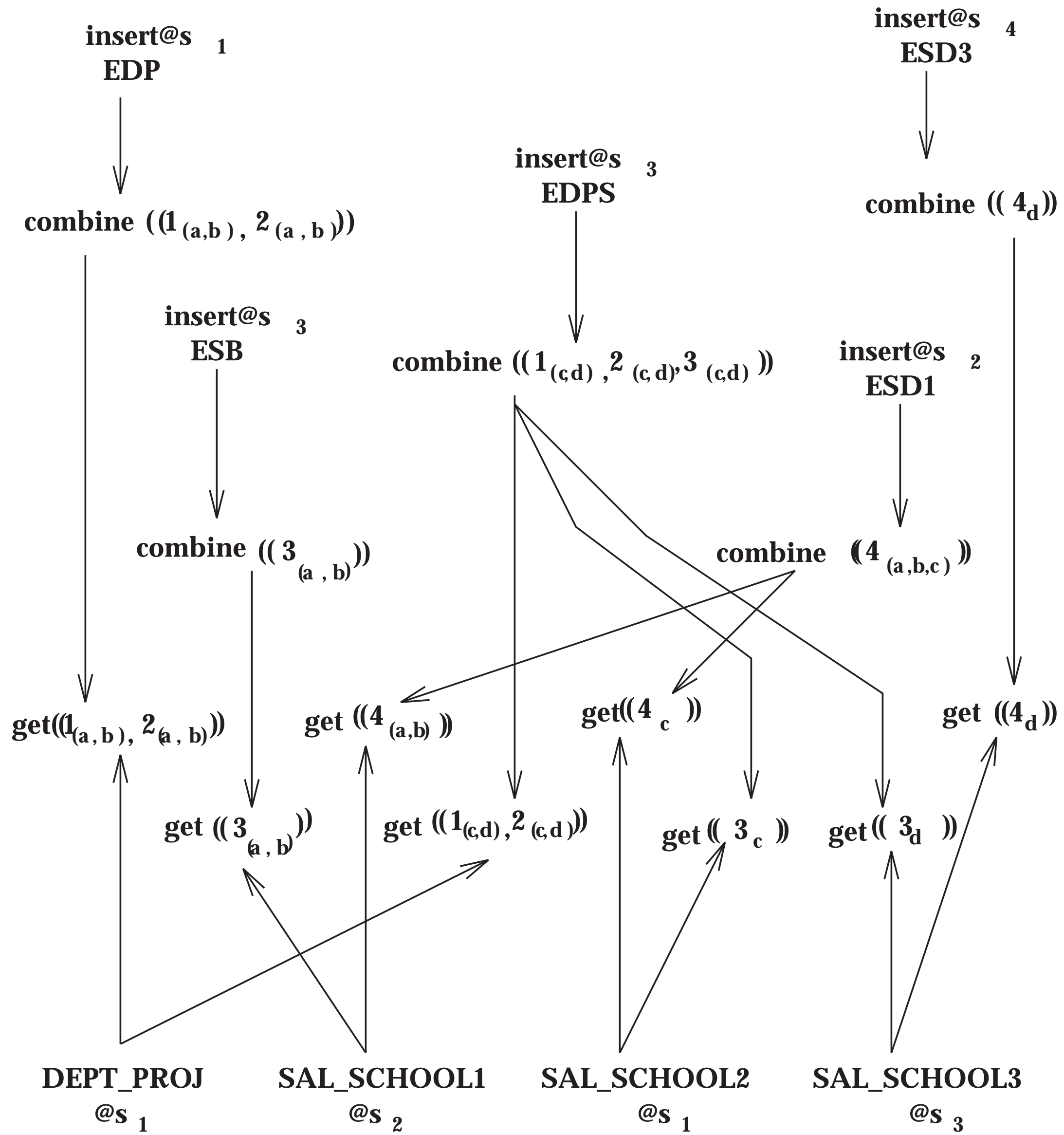


Figure 26: Combined optimized query graph

WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20;

INSERT INTO $TempTable_2$ Emp#, Name, City, Dept#, Proj#

SELECT Emp#, Name, City, Dept#, Proj#

FROM DEPT_PROJ

WHERE Dept# $\geq$ 30 AND 20 $\leq$ Dept# $<$ 30;

INSERT INTO $TempTable_5$ Emp#, Salary, Bonus

SELECT Emp#, Salary, Bonus

FROM SAL_SCHOOL2

WHERE 20 $\leq$ Dept# $<$ 30;

INSERT INTO $TempTable_6$ Emp#, School, Degree

SELECT Emp#, School, Degree

FROM SAL_SCHOOL2

WHERE 20 $\leq$ Dept# $<$ 30;

end{concurrent}

send_to(s_3) $TempTable_2$

send_to(s_3) $TempTable_6$

send_to(s_2) $TempTable_5$

At site s_2:

begin{concurrent}

INSERT INTO $TempTable_3$ Emp#, School, Degree

SELECT Emp#, School, Degree

FROM SAL_SCHOOL1

WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20;

INSERT INTO $TempTable_4$ Emp#, Salary, Bonus

SELECT Emp#, Salary, Bonus

FROM SAL_SCHOOL1

WHERE Dept# $<$ 10 AND 10 $\leq$ Dept# $<$ 20;

end{concurrent}

send_to(s_3) $TempTable_4$

At site s_3:

begin{concurrent}

INSERT INTO $TempTable_7$ Emp#, Salary, Bonus

SELECT Emp#, Salary, Bonus

FROM SAL_SCHOOL3
WHERE Dept# $\geq$ 30;
INSERT INTO $TempTable_8$ Emp#, School, Degree
SELECT Emp#, School, Degree
FROM SAL_SCHOOL3
WHERE Dept# $\geq$ 30;
end{concurrent}
send_to(s_4) $TempTable_8$

Following are the set of SQL statements that need to be executed at each of the sites where the fragments of the new distributed database design need to be materialized. Note that the actual relation names and column names are used instead of the representations of the vertical and horizontal fragments.

At site s_1:
INSERT INTO EDP Emp#, Name, City, Dept#, Proj#
SELECT Emp#, Name, City, Dept#, Proj#
FROM $TempTable_1$;
At site s_2:
INSERT INTO ESD1 Emp#, School, Degree
SELECT Emp#, School, Degree
FROM $TempTable_3$, $TempTable_6$
WHERE $TempTable_3$.Emp# = $TempTable_6$.Emp#;
At site s_3:
INSERT INTO EDPS Emp#, Name, City, Proj#, Dept#, Salary, Bonus
SELECT Emp#, Name, City, Proj#, Dept#, Salary, Bonus
FROM $TempTable_5$, $TempTable_7$
WHERE $TempTable_5$.Emp# = $TempTable_7$.Emp#
UNION
SELECT Emp#, Name, City, Proj#, Dept#, Salary, Bonus
FROM $TempTable_2$;

INSERT INTO ESB Emp#, Salary, Bonus
SELECT Emp#, Salary, Bonus
FROM $TempTable_4$;

At site s_4:

INSERT INTO ESD2 Emp#, School, Degree
SELECT Emp#, School, Degree
FROM $TempTable_8$;

In the next section the algorithm to optimize the set of SQL statements to materialize the new design is given.

6.4 MQO Algorithm to Materialize Distributed Database Design

The algorithm consists of the following steps:

MQO_materialization(SQL_in, SQL_out)

Input: SQL_in a set of SQL statements generated by the query generator algorithm.

Output: SQL_out optimized set of SQL statements to be executed at various sites of the distributed database environment.

1. Select a site s_* in the distributed database environment to be the coordinator site for the MQO. All the query modification steps will be performed at the site s_*.

2. For each $f' \in F'$, the **Query_Generator** generates the following SQL statement. $c(r(f'))$ denotes the columns and $p(r(f'))$ denotes the disjunction of the predicates specified in the representation of the fragment f'.

INSERT INTO f' $c(r(f'))$
SELECT $c(r(f'))$
FROM R
WHERE $p(r(f'))$;

---

3. Replace relation R by the fragments in the minimal cover $m(f')$ generating the following modified SQL statement. $\phi(f')$ gives the locations of the sites where f' is to be allocated

---

insert@$\phi(f')f'$
SELECT $c(r(f'))$
FROM $m(f')$
WHERE $p(r(f'))$;

---

4. Rewrite the SQL statement using the fragment representations, *combine* function, and *get* function. $\{f@\phi(f) \mid f \in m(f')\}$ lists the fragments in the minimal cover of the fragment f', Then the new generated query expression is:

---

insert@$\phi(f')$ f'
$combine(r(f'))$
$get(r(f'))$
from $\{f@\phi(f) \mid f \in m(f')\}$

---

5. Propagate the *get* function to each of the fragments in the $int(f')$, thus rewriting the above query expression gives:

---

insert@$\phi(f')$ f'
$combine(r(f'))$
for each $f \in m(f')$ generate:
    $get(r(f'))$
    from $f@\phi(f)$

---

6. Modify the grid cells that need to be retrieved from each of the fragments of the current fragmentation scheme, giving the new modified query expression as:

---

insert@$\phi(f')$ f'

$combine(r(f'))$

for each $f \in m(f')$ generate:

$get(r(f') \cap r(f))$

from $f@\phi(f)$

---

7. Generate two sets of statements one for each fragment of the current fragmentation scheme and one for each fragment of the new fragmentation scheme. This done by considering the *minimal cover* of the fragment f' and the *intersect* of the fragment f.

---

for each $f \in F$ generate

for each $f' \in int(f)$

$get(r(f) \cap r(f')) \wedge send_to(\phi(f'))$ as $TempTable_{(f,f')}$

from $f@\phi(f)$

---

for each $f' \in F'$ generate

$insert@\phi(f')f'$

$combine_{f \in m(f')}(TempTable_{(f,f')})$

---

8. Generate the set of optimized SQL statements that need to be generated simultaneously at each site of the current and new distributed database designs are:

---

for each site $s \in S$ do:

define $frg(s) = \{f \mid s \in \phi(f) \& (f, \phi(f)) \in F\}$

$begin\{concurrent\}$

for each $f \in frg(s)$

for each $f' \in int(f)$ generate

INSERT INTO $TempTable_{(f,f')}c(r(f) \cap r(f'))$

SELECT $c(r(f) \cap r(f'))$

FROM f

WHERE $p(r(f) \cap r(f'));$

$end\{concurrent\}$

for each $f \in frg(s)$

for each $f' \in int(f)$

if $\phi(f) \neq \phi(f')$ generate

$to_send(\phi(f'), TempTable_{(f,f')})$

---

for each site $s \in S$ do:

define $frg'(s) = \{f' \mid s \in \phi(f') \& (f', \phi(f') \in F'\}$

for each $f' \in frg'(s)$ generate

$form_sql(f')$

The function $form_sql(f')$ generates the SQL statement to materialize the fragment f'. Note that all the fragments in the current distributed database design and the new distributed database design are regular fragments. Also, all the elements of the set $\{r(f) \cap r(f') \mid f \in m(f')\}$ are regular. Note that the fragments represented by grid cells $r(f) \cap r(f')$ define the $TempTable_{(f,f')}$. These temporary tables are created in the previous step. Hence to materialize fragment f' we need to merge these temporary tables. There are two ways of merging tables, one is *join* and the other is *union* of tables. So in order to automate this merging process, the correct order of these merging operations needs to be generated. Let $r(f') = \{\alpha_{1A}, \alpha_{2A}, \alpha_{nA}\}$,and A be set of horizontal fragments defined by predicates $a, b, \ldots, s$. Then the SQL statement to materialize the fragment f' is:

$form_sql(f')$

INSERT INTO f' $c(f')$

SELECT $c(f')$

FROM $TempTable_{(f,f')}\{f \mid a \in r(f)\ and\ f \in m(f')\}$
WHERE p(a) and join($TempTable_{(f,f')}$)$\{f \mid a \in r(f)\ and\ f \in m(f')\}$
UNION
SELECT $c(f')$
FROM $TempTable_{(f,f')}\{f \mid b \in r(f)\ and\ f \in m(f')\}$
WHERE p(b) and join($TempTable_{(f,f')}$)$\{f \mid b \in r(f)\ and\ f \in m(f')\}$
UNION
⋮
UNION
SELECT $c(f')$
FROM $TempTable_{(f,f')}\{f \mid s \in r(f)\ and\ f \in m(f')\}$
WHERE p(s) and join($TempTable_{(f,f')}$)$\{f \mid s \in r(f)\ and\ f \in m(f')\}$;

The function *join* takes as its input a set of tables and generates a set of join conditions to join these tables. For joining k tables, $(k-1)$ join conditions are generated. The above algorithm first joins all the tables to generate the complete horizontal fragments of the fragment f' and then it performs a UNION of all these horizontal fragments. To form a horizontal fragment it joins the temporary tables that have this horizontal fragment included in them and selects only those tuples that satisfy this horizontal fragment. This is a brute force method to materialize the fragment f', but any other method would be much more complex as different cases need to be considered. The SQL statement to materialize a fragment would depend on the cardinality, shape and size of the elements of the minimal cover of a fragment. If the minimal cover has only one element, then the above SQL statement is very simple. If it has two elements, then the statement uses either a join or a union. But as the number of elements in the minimal cover increase, it is not always simple to correctly specify the right order of merging these temporary tables. This is the reason that the above approach of first constructing all the horizontal fragments and then forming a union of them was selected. In some specific cases a simpler SQL statement can be constructed, but in order to provide a general and correct solution, that approach was not taken. Moreover, the local database system may use the query optimizer to optimize this SQL statement so as to execute it efficiently.

9. The above set of modified SQL statements are then sent from site s_* to all other respective sites to be initiated. Once these SQL statements are completely executed, the new distributed database design is completely materialized.

The above set of SQL statements can be generated by an extensible query optimizer like VOLCANO [Goe93] and executed at various sites of the distributed database system. This would require no change in the query decomposition and optimizer modules of the distributed database system.

6.5 Performance Improvement

The multiple query optimization that is performed assures the performance of the query generator method to be the same as that of the operator method. The main difference between the two was that operator would generate all the sub-fragments defined by $int(f)$ of a fragment $f \in F$ by reading in the relation only once. Whereas in case of the query generator method, if there were no suitable indexes, then the relation would be scanned completely more than once to generate all the sub-fragments. This implied that the query generator method can perform worse than the operator method.

In this multiple query optimization algorithm all the sub-fragments of a fragment are generated at once by executing the first part of the Step 8. This would essentially mean that the relation would be read only once to generate all the sub-fragments. Of course, this depends on the implementation of the relational database system. But most commercial database systems use buffering schemes that enable a relation to be read only once. This is the reason that all the temporary tables are generated simultaneously. As a page of a relation read can be written into pages of all the temporary tables that require that data at once. This would require an allotment of some buffers to all the temporary tables. With the amount of main memory that current systems have, it is feasible to do this. Similarly when merging the temporary tables to form the fragment of the new distributed database design, all the temporary tables will be read only once.

Thus this optimized query generator algorithm's performance in terms of number of disk accesses will be the same as that of the operator method.

6.6 Summary

In this chapter the queries generated by the **query generator** were analyzed and optimized to generate a modified set of queries to be executed at each of the sites of the distributed database environment so as to materialize the new distributed database design. The performance of this optimized set of queries was argued to be that of the operator method. Note that operator method consists of three phases *Divide*, *Relocate* and *Conquer*, and these modified set of SQL statements also materialize by first generating the temporary tables, second, moving them to different sites (if necessary), and third, merging them to form the new fragments. This optimized set of SQL statements emulate the operator method and use the distributed database system to materialize the new distributed database design. The operator method can still be used if the distributed database system cannot handle redistribution of large sets of data and if the materialization has to be done off line or incrementally.

CHAPTER 7

AVERAGE TRANSACTION RESPONSE TIME ESTIMATION

7.1 General Comments

The average transaction response time is used to evaluate the efficiency of two distributed database designs. It is also used as an estimate for the cost of using a particular distributed database design for a class of applications. This cost value is used by the Markovian decision analysis technique to generate the optimal policy for the distributed database redesign methodology in the application processing center scenario. It is very difficult to come up with a general analytical model that takes into consideration different interdependent characteristics of the distributed database environment. Hence we present a simulation model which is general enough to support different characteristics of the distributed database environment. This model is an extension of the model proposed by [YDR+85].

The method that we are using to estimate the average transaction response time is one of the many that can be used. The other methods include different simulation models (like [CL88]), different analytical models, and actual measurements on the distributed database environment. Our goal here is to provide a method to estimate the average transaction response time. In the formulation of the cost factors in the Markov Decision Analysis problem we can use any of the above mentioned methods.

Moreover, the simulation model that we are using in this thesis enables us to evaluate the affect of change in various parameters on the average transaction response time. This chapter is organized as follows: Section 7.2 discusses the distributed database environment, Section 7.3 on transaction characteristics, Section 7.4 on estimating probability of contention, Sections 7.5 and 7.6 on the simulation model and Section 7.7 on the results of the experiments conducted.

7.2 The Distributed Database Environment

The distributed database environment that we are considering in this thesis consists of a set of computers connected by a network. Each of the computers has a relational database system that can coordinate as a distributed database system. Following are the characteristics of the distributed database environment:

1. All the computers are connected by means of a network with a certain data transfer rate.

2. Each computer system has a database management system that manages a number of locks.

3. Each computer has a distributed database system that enables access to data from any other site (computer) in the distributed database environment.

7.3 Transaction Characteristics

A transaction is an atomic unit of consistent, reliable and durable computation. In distributed database environment there are two kinds of transactions: *local transactions* and *distributed transactions*. A local transaction accesses data only from the site where it was initiated. A distributed transaction accesses data from more than one site. An application consists of a set of transactions. It is the applications that are executed in the distributed database environment. An execution of a transaction consists of loading the transaction initiation program, accessing and processing the data from the database, and finally committing or aborting. Figure 27 shows the phases of transactions described below.

A local transaction has the following phases during its lifetime:

Initiation The program initiating the local transaction has to be loaded into the main memory and then executed. This incurs some disk I/O's and some CPU processing.

Database access The transaction then goes through a set of actions. An action being "a request for a lock", "release a lock", and "data access". The data access here is from the database and hence the time for it will be different from the time required to do a disk I/O. There is some "CPU processing" between any two successive actions.

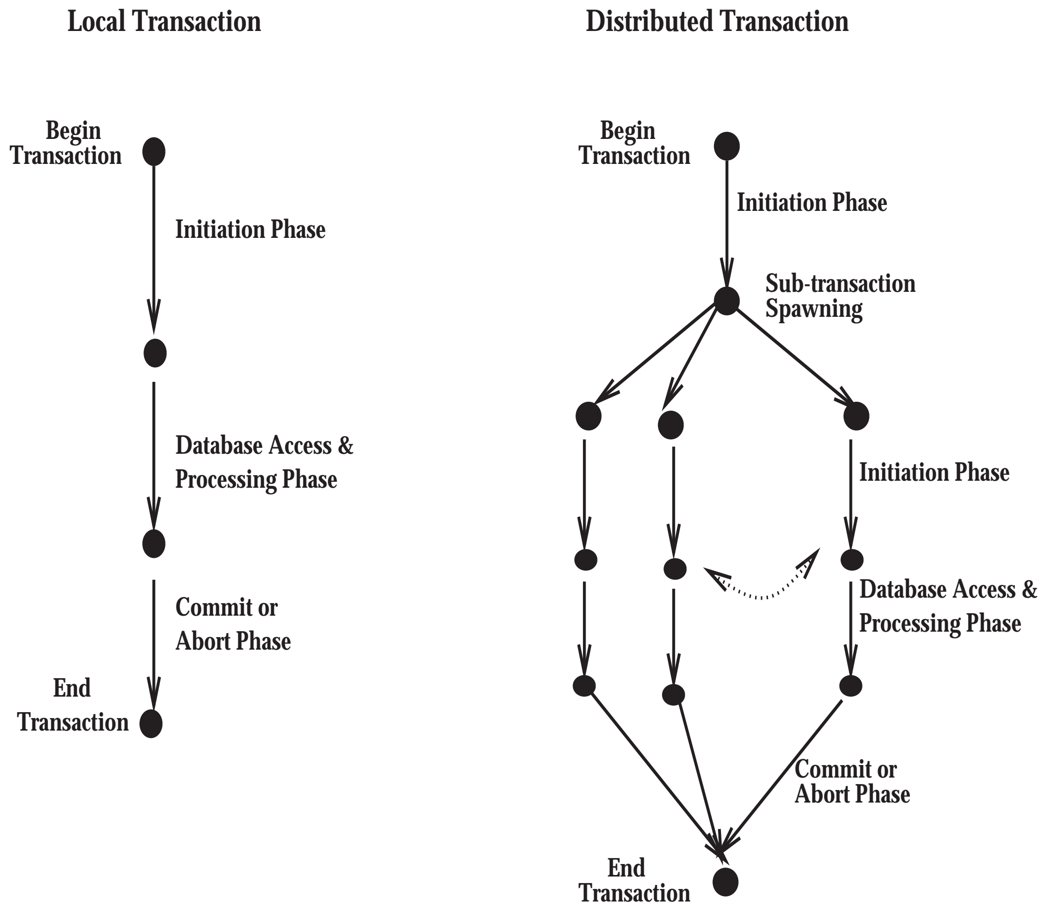


Figure 27: Local and Distributed Transaction Phases

Commit or Abort The transaction then either commits by writing a log record or aborts. A commit incurs writing a log record to stable storage and then releasing the locks held.

A distributed transaction has the following phases during its lifetime:

Initiation The program initiating the distributed transaction has to be loaded into the main memory and then executed. This incurs some disk I/O's and some CPU processing.

Spawning The distributed transaction spawns a number of sub-transactions as local transactions at different sites. This incurs some communication delay to initiate sub-transactions at different sites. These local transactions access and process the data

from the database as given above. Once the local sub transactions of the distributed transaction finish their processing, there is some data that is transferred to the site where the distributed transaction is initiated.

Commit or Abort Once the local sub-transactions finish processing the data accessed from the local database, the distributed transaction initiates a two phase commit protocol. The distributed and its local sub-transactions commit or abort based on the outcome of the commit protocol.

On any given computer of the distributed database environment both local and distributed transactions are executed concurrently. Hence they may contend for the locks on the data item they access. In case a transaction A contends for a lock with transaction B, then transaction A has to wait until either transaction B either commits or aborts. The wait time for a transaction A during contention is directly proportional to the lock hold time of the transaction B.

In order to take into consideration the affect of contention on the transaction response time, we can explicitly model data contention or estimate the probability of contention for each lock. In this thesis the latter choice is opted for as this makes the simulation model for transaction processing in distributed database environment simple. This will also reduce the amount of time taken to run the simulation. If the data contention is modeled explicitly with about 100K data items, and the transactions access these data items randomly, then there is a lot of bookkeeping that needs to be done so as to keep track of data contention. This bookkeeping which is required for each transaction slows down the simulation.

The effect of contention on transaction response time is cyclic in nature. That is, the contention increases the transaction response time, which increases average lock hold time, which in turn increases the probability of contention. This cyclic affect of contention needs to be incorporated into the simulation model. If the distributed database environment being simulated is stable, then the system stabilizes to a value of probability of contention. We shall present a mechanism to estimate the probability of contention and will use it in the simulation.

7.4 Estimating the Probability of Contention

Let λ_l and λ_d be the arrival rate of local and distributed transactions at each site respectively. Let N_l and N_{lt} be the number of locks on an average requested by local transactions

and local sub-transactions of distributed transactions respectively. Let P_l be the probability that a transaction being executed at a site is a local transaction. Let P_{lt} be the probability that a transaction being executed at a site is a local sub transaction of a distributed transaction. Let each distributed transaction fork into n_t sub-transactions on an average, then the arrival rate of local sub transaction of distributed transaction at each site as $n_t \times \lambda_d$. Let the number of sites in the distributed database environment be N_{DS}, and let the number of locks in the distributed database environment be $LSPACE$. Then the probability of contention of a local transaction with a local transaction denoted by $P^{l/l}_{CONT}$ is given by:

$$P^{l/l}_{CONT} = \frac{(\lambda_l + n_t\lambda_d)N_l P_l \frac{LocalLockHoldTime}{2}}{LSPACE/N_{DS}} \tag{7}$$

The numerator gives us the average number of transactions held by local transactions estimated by the product of average number of locks requested by local transactions and the average lock holding time of local transactions as given by Little's Law. Each computer system of the distributed database environment is the master of $LSPACE/N_{DS}$ locks. The average lock holding time is given by $LocalLockHoldTime/2$ assuming that the transaction acquires locks uniformly over the time it holds the locks. The above formula is the adaptation of the technique from [CDY90] to the distributed relational database environment.

Similarly, the other probabilities of contentions are given by:

$$P^{l/lt}_{CONT} = \frac{(\lambda_l + n_t\lambda_d)N_l P_l \frac{DistLockHoldTime}{2}}{LSPACE/N_{DS}} \tag{8}$$

$$P^{lt/l}_{CONT} = \frac{(\lambda_l + n_t\lambda_d)N_{lt} P_{lt} \frac{LocalLockHoldTime}{2}}{LSPACE/N_{DS}} \tag{9}$$

and

$$P^{lt/lt}_{CONT} = \frac{(\lambda_l + n_t\lambda_d)N_{lt} P_{lt} \frac{DistLockHoldTime}{2}}{LSPACE/N_{DS}} \tag{10}$$

Note that the probabilities of contention of a transaction is directly proportional to the lock hold time of the transactions they contend with. In the simulation, the lock hold times are actually measured and averaged over a number of transactions. These values are used in the above formulae to estimate the probabilities of contention.

7.5 A Model For The Locking Mechanism

The model for locking mechanism developed by [YDR+85] is extended to model the contention in a distributed relational database transaction processing environment. Each transaction that enters the distributed database environment is either a distributed transaction or a local transaction. The percentage of distributed transactions is fixed for each simulation run. As described in Section 7.3, a distributed transaction under-goes initiation, sub-transaction spawning and commit or abort phases. The initiation phase consists of some CPU processing with interleaved disk accesses. The sub-transaction spawning phase incurs some CPU overhead and communication delay to spawn a number of sub-transactions at different computer systems of the distributed database environment. Each of these sub-transactions like local transactions go through the initiation phase and the database access and processing phase. It is while accessing the tuple from the database that the transaction may have contention with other transactions. Figure 28 shows a model of lock manager being used in our distributed database environment.

The lock manager is the core of the database system that manages the access to the data. After the initiation phase, each local transaction or local sub-transaction of the distributed transaction enters the phase of database access and processing which is governed by the lock manager. Each transaction has a number of predefined actions. These actions are enumerated as number of data accesses, number of locks requested, number of unlocks, and transaction termination. In our approach, we use the wait lock model [YDR+85], that is, the CPU is not released when a lock request is made. When a lock request is made, the CPU waits for the response of lock manager, if the lock is "granted" the transaction continues, otherwise there is a "contention" resulting in the CPU to switch to another transaction by paying the task switch overhead.

A contention results in the transaction to wait for the lock to be released. The database tuples are not explicitly modeled. The probability of contention derived in Section 7.4 will be used to decide whether a transaction must wait. The transaction releases some of its locks during its database access and processing phase, but releases all its remaining locks at the commit or abort phase. The transaction does not wait for the response for its unlock request (which is assumed to take negligible amount of time) from the lock manager and continues with its actions. The local transaction (or local sub-transaction of a distributed transaction) undergoes a number of data accesses from the database during which locks are

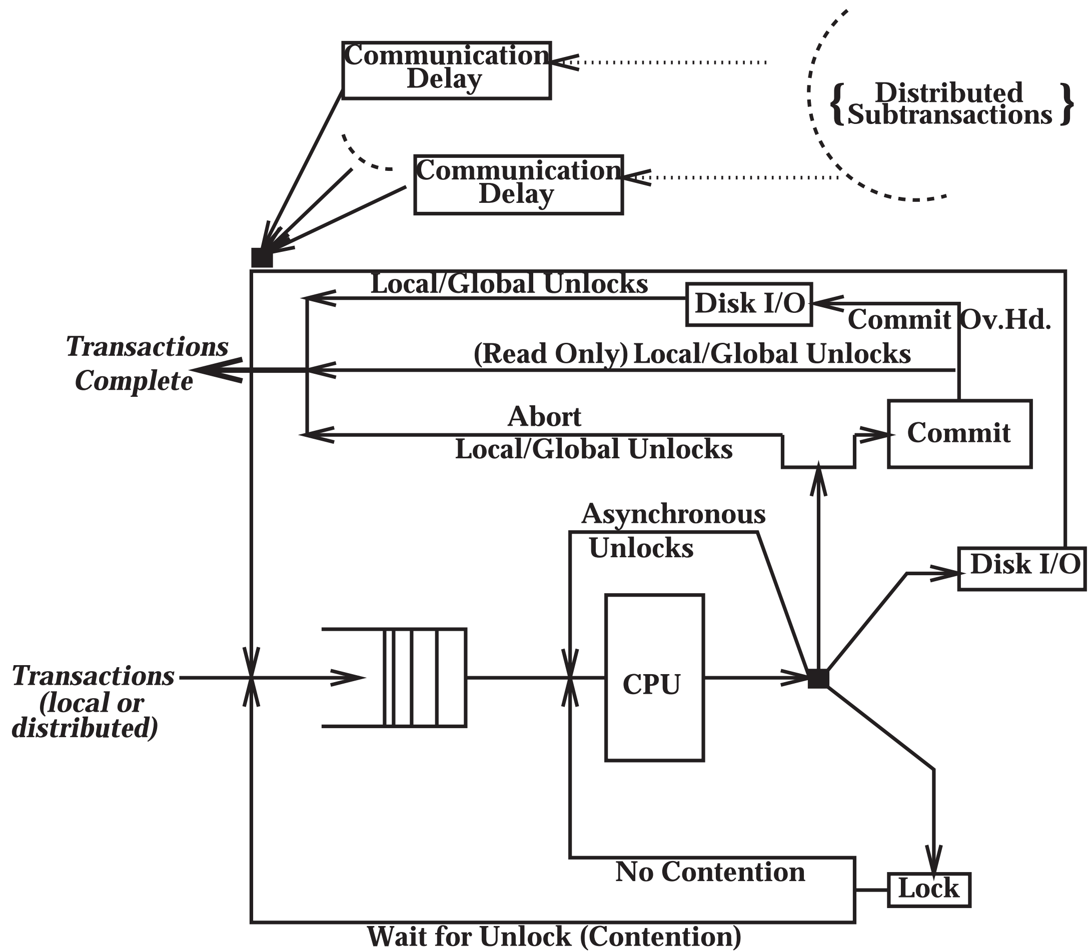


Figure 28: Simulation Model For Lock Contention

held on the data. This has to be explicitly modeled because the wait time due to contention is proportional to the part of the response time during which the locks are held.

The local transaction commit consists of writing into the stable storage a write-ahead log so that the transaction can be recovered in case of failure of the system. Once the write-ahead log is written, any locks held by the transaction are released. For a distributed transaction a two-phase commit protocol is used.

7.6 Simulation Model

The distributed transaction processing system outlined in the previous section was simulated using an event driven simulation model. An open queuing model with Poisson transaction

Table 5: Workload Parameters

Parameter	Value
Trans Rate	2 to 20 Trans./sec/system
Percentage of Distributed Trans	10 to 90
Avg Num of Sub-trans/Distributed trans	3
Locks/Local Trans	30
Unlocks/Local Trans	30
Unlocks at purge for Local Trans	4
Initiation I/O Local Trans	5
Locks/Local Sub-Trans	10
Unlocks/Local Sub-Trans	10
Unlocks at purge for Local Sub-Trans	2
Data accesses for Local Trans	30
Data accesses for Local Sub-Trans	10
Trans path length for Local trans	313000 ins.
Trans path length for Local Sub-Trans	248000 ins.
Avg Communication Delay for Distributed Trans	0.0 Sec to 1.0 Sec

arrival process is assumed. This is reasonable for modeling the system throughput versus response time characteristic. For a large number of transactions with response time of few seconds, the net transaction arrival process is independent of the response time.

There are two kinds of parameters that are considered in the simulation model, namely, workload and system. The workload parameters characterize the behavior of the transaction in terms of transaction path length, number of locks, unlocks and data accesses. The system parameters are much more static parameters of the distributed database environment like CPU MIPS, the lock manager response time, the data access time. The overheads required to carry out various actions (like lock, unlock, commit, etc.) are also modeled. The frequency of the overheads and actions depends on the workload parameters. Table 5 shows the workload parameters and Table 6 shows the system parameters used in our simulation runs. These parameter values have been adapted from those used by [YDR+85] to model concurrency control in multi-system environments, and by [CDY90] in their analysis of replication of distributed database systems.

A transaction that arrives at a system is designated as a local or distributed transaction. Then after the initiation phase, the local transaction enters the lock holding phase (i.e.

Table 6: System Parameters

Parameter	Value
Num of Systems N_{DS}	10
Initial disk access time	35 Mill-Sec
Lock Service time	100 Micro-Sec
Num of Locks in Distributed Database System	100000

database access and commit or abort phases together). In this phase the different actions of the transactions are modeled as multiple class jobs with the CPU as the dispatcher queue.

The sum of the average number of locks, unlocks, database I/O's and commit, will be referred to as the number of events per transaction. The total transaction path length (average number of instructions executed per transaction) is divided by the number of events to give the average number of instructions executed between two consecutive events. In the model, a transaction runs on a CPU for a time corresponding to the instructions between events. The event after this is modeled as a lock, unlock, data access or a commit action with probability computed as the ratio of the average number of each of these events per transaction to the average number of events per transaction. This leads to a geometrically distributed transaction path length with correct mean.

The modeling of a lock request is an important aspect of the simulation. After the transaction requests the CPU for a lock, it is routed to the lock manager. Once the lock manager grants the lock after certain delay, the transaction joins the CPU queue for the next inter event path length. If the transaction has a contention for the lock then it is handled as given below.

The simulation models the cyclic nature of the effect of probability of contention on transaction response time and vice versa by using a feedback process. As mentioned in Section 7.4, the probability of contention is proportional to lock request rate and average lock hold time. The initial estimate of probability of contention is based on no contention. During the simulation the probability of contention is calculated based on average transaction response time of previous few thousand transactions. This new calculated probability of contention is used for the next few thousand transactions. This is done iteratively till there is no appreciable change in probability of contention values for two consecutive runs. The commit of a local transaction is modeled as additional disk I/O to update the log in the

stable storage. In case of distributed transaction the commit is modeled as a disk I/O at each of the sites participating in the commit protocol to update the log in the stable storage. Additionally, a 1/4th of the communication delay is used as a measure of communication delay for two-phase commit protocol.

The spawning of local sub-transactions involves a CPU overhead and a communication delay for initiating a local sub-transactions on different systems. The communication delay for data and message passing between the distributed transaction and local sub-transactions is also based on a single communication delay value. The total communication delay starting from initiating the local sub-transactions till the start of two-phase commit protocol is modeled as twice the average communication delay for distributed transaction. This is because there is communication delay involved in spawning the local sub transactions of a distributed transaction, and data transfer after the local sub transactions have processing the data. The actual amount of time spent on data transfer can be estimated from the trace driven simulation of the distributed transactions.

7.7 Experiments

Given a set of system and workload parameter values, the above simulation can be run to calculate the average transaction response time. This enables us to compare two distributed database designs by comparing the average transaction response times when they are used to execute a class of applications. This simulation model can also be used to analyze the effect of change in some of the parameters on the average transaction response time. The arrival rate for transactions is that of both local and distributed transactions together.

7.7.1 Effect of Data Access Time

We have been using data access instead of disk I/O in this thesis. Disk I/O is the time taken to access a page or a set of pages from the stable storage (mostly disk). The data access time includes the amount of CPU processing involved in accessing a page of tuples from the relation. Thus an increase in data access time is caused by executing the data access routines on a computer with smaller MIPS rating. Also increase in page size increases the data access time. We use the transaction arrival rate as 5 transactions/sec/system, and average communication delay as 0.5 seconds. The data access time is varied from 0.06 seconds to 0.09 seconds. Initially, we have the average transaction response time increase

as the percentage of distributed transactions increase (for the case where data access time equals 0.06 Sec). But as the data access time increases, the average transaction response time first decreases and then increases as the percentage of distributed transactions increase. This implies that there is an optimal percentage of distributed transactions which gives the minimum average transaction response time. This is percentage is about 25 when data access time is 0.08 second and about 37 when the data access time is 0.09 second.

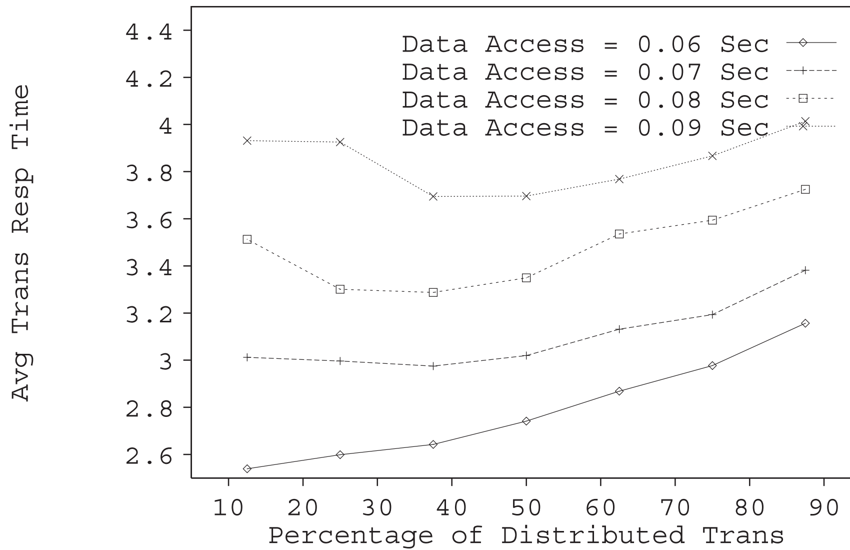


Figure 29: Effect of Data Access Time on Average Transaction Response Time

This is because while the distributed transaction time is waiting for the communication delay, the local transaction can access the data and process it. At lower percentage of local transactions, there are not enough local transactions to take advantage of this scenario. But as the percentage of local transactions increase more transactions can take advantage of this scenario. But as the percentage of local transactions exceed the optimal point, they start having contention among themselves, and thus increasing average transaction response time. As the data access time increases, the optimal percentage point for distributed transactions increase, this is because the local transactions start holding locks for longer period thus having higher probability of contention which results in the increase in the average transaction response time.

7.7.2 Effect of Arrival Rate of Transactions

In this subsection the effect of change on the arrival rate on average transaction response time is studied. The increase in arrival rate increases the contention for data access, thus increasing the average transaction response time. This depends on the average number of data accesses made per local transaction, and per local sub-transaction of the distributed transaction. In our first experiment the communication delay was 1 second, and the data access time was 0.1 seconds. The local transaction on an average perform 30 data accesses and the local sub-transactions of the distributed transaction on an average perform 10 data accesses.

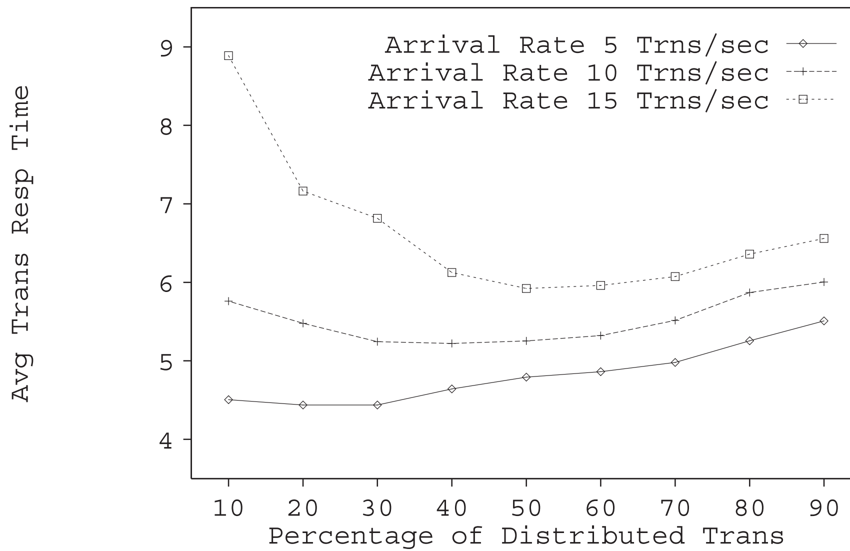


Figure 30: Effect of Arrival Rate on Average Transaction Response Time: I

As the arrival rate increases from 5 trans/sec till 15 trans/sec, the average transaction response time increases. But as the percentage of the distributed transactions was varied, for low arrival rates there is not much contention and hence larger percentage of local transactions gives the best average transaction response time. This percentage is about 70. But as the arrival rate increases to 10 transactions per second the contention among local transactions increase and thus increasing the average transaction response time. This is the reason that larger percentage of distributed transactions give the best average transaction

response time. By doing this, the contention among the local transactions is reduced. The percentage of distributed transactions that give this average transaction response time is 40.

When the arrival rate of transactions is 15, the contention among local transactions is very high which increases the average transaction response time. When the percentage of local transaction is very large, the average transaction response time is also large. But as the percentage of local transactions decreases the contention decreases and the average transaction response time decreases. The best average transaction response time occurs when the percentage of the distributed transactions is about 50. Note that as the percentage of distributed transactions increases, the contention among them increases giving rise to an increase in average transaction response time.

An increase in arrival rate increases the probability of contention, thus increasing the average transaction response time. The increase in probability of contention depends on the number of data accesses. If the number of data accesses for local transactions are more than that of local sub-transactions of the distributed transactions, then it is better to have some percentage of distributed transactions and this percentage increases as the arrival rate increases.

The case where the number of data accesses for local transactions is comparable to the number of data accesses for sub-transactions of the distributed transaction is illustrated in Figure 31. The number of data accesses for local transaction is 15/transaction, and that for a sub-transaction of a distributed transaction is 12/sub-transaction. For this case the average transaction response time for a distributed transaction is persistently higher than that of a local transaction. Hence the optimal percentage of distributed transactions for the least average transaction response time is zero. Also note that increasing the arrival rate of transactions to 15 transactions/sec, the optimal percentage of the distributed transactions remains at zero because the number of data accesses for local transactions is comparable to the number of data accesses for sub-transaction of a distributed transaction.

7.7.3 Evaluation of the Optimal Percentage of Distributed Transactions

The average transaction response time in the distributed database environment was simulated for a range of data access times from 0.02 seconds to 0.09 seconds and communication delays from zero seconds till 2.5 seconds with arrival rate of 10 transactions per second.

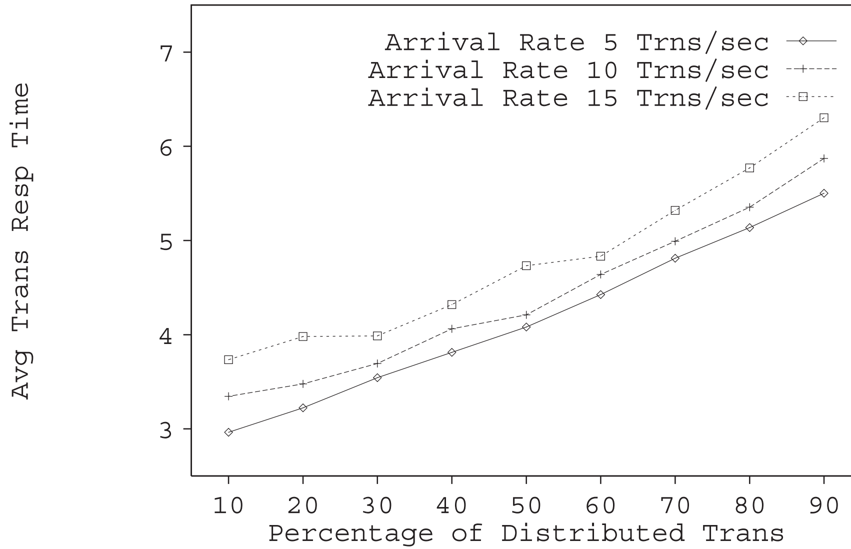


Figure 31: Effect of Arrival Rate on Average Transaction Response Time: II

For each simulation the data access time and the communication delay were fixed and the percentage of distributed transactions was varied from 10 to 90. The average number of data accesses for each local transaction was 30, and for each sub-transaction of a distributed transaction was 10. The percentage of distributed transactions for each run that gives the least average transaction response time was determined. The data access time was fixed and the communication delay was varied from zero seconds till 2.5 seconds. Below certain communication delay it was better to have all distributed transactions as shown in the Figure 32 by the graph line `Below All Distributed`. This means that the time spent on data transfer is less compared to the amount of time spent on data access for a distributed transaction. Therefore, all the transactions in the distributed database environment need to be distributed to give the least average transaction response time. Similarly, above a certain communication delay it was better to have all local transactions as illustrated by the graph line `Above All Local` in Figure 32. That is the amount of time spent on data transfer is more than the about of time spent on data access. Therefore, all transactions executed in the distributed database environment need to be local to give the least average transaction response time. This graph clearly shows the trade off between the amount of time a distributed transaction can spend on data transfer and data access.

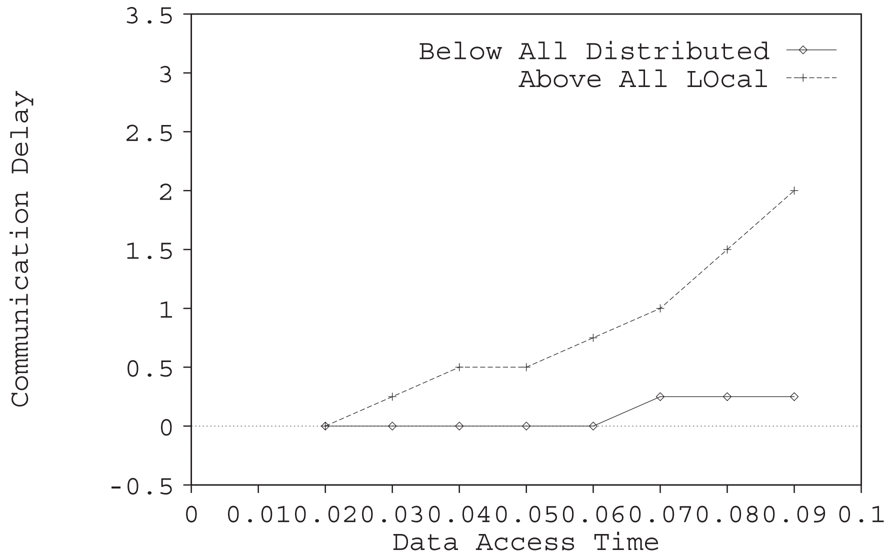


Figure 32: Evaluation of Optimal Percentage Points with Arrival Rate 10 trns/sec

For the communication delay in the range between the two graph lines, it is better to have a certain percentage of distributed transactions for the least average transaction response time. Note that as the data access time increases, the range of communication delay over which it is better to have some percentage of distributed transactions increases. For example, when it is 0.05 seconds the range is from 0.0 seconds till 0.50 seconds. whereas when the data access time is 0.08 seconds, the range is from 0.25 seconds till 1.50 seconds, That is, if the average communication delay in the distributed database environment is below 0.25 seconds, it is better to have all distributed transactions, if it is above 1.50 it is better to have all local transactions, and if it is in the range [0.25:1.50] then it is better to have some percentage of distributed transactions. Note that this range increases as the data access time increases. Mainly, it is the communication delay value for `Above All Local` that increases exponentially as the data access time increases. This is because the communication delay for the distributed transaction is compensated by the delay due to data contention for the local transactions. Only when the delay due to data contention for local transactions offsets the effect of communication delay for a distributed transaction, that it is better to have all local transactions. As data access time increases, the delay due to data contention for local transactions increases, and it takes higher communication delay

to overcome this, so as to make it better to have all local transactions.

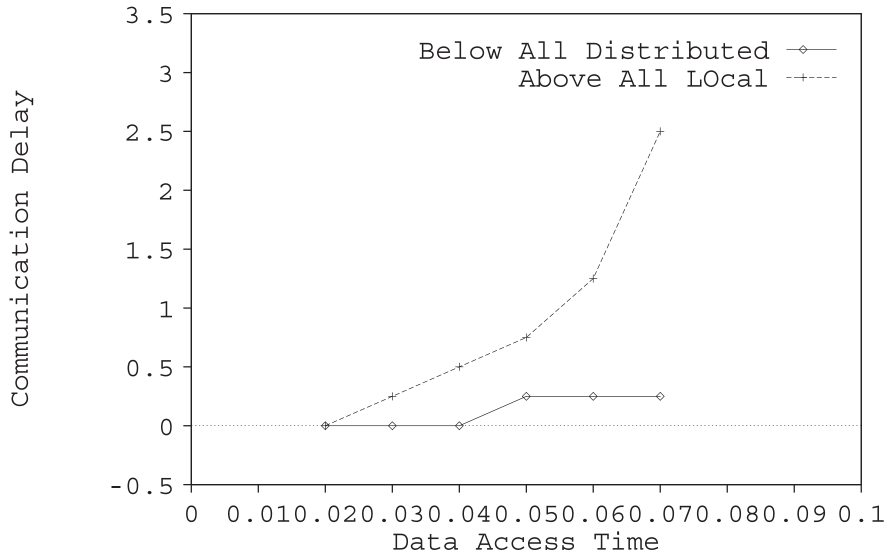


Figure 33: Evaluation of Optimal Percentage Points with Arrival Rate 15 trns/sec

This increase in the range is much more prominent when the arrival rate of transactions is increased to 15 transactions/second as shown in the graph of Figure 33. For this case when the data access time is 0.070 seconds, the range for communication delay is from 0.25 seconds till 2.5 seconds. This is because the higher arrival rate increases the delay due to data contention for local transactions much more than that for distributed transactions. For high values of data access times and communication delay, the distributed database environment was found to be unstable. Hence, for the case of the arrival rate of 10 transactions/second, data access time until 0.09 seconds was considered, and when the arrival rate was 15 transactions/second, data accesses time until 0.07 seconds was considered. But the interesting observation is that for a wide range of communication delay it is beneficial to have some percentage of distributed transactions when the arrival rate is 10 or 15 transactions/sec and data access time is greater than 0.060 seconds.

The simulation results in Figures 32 and 33 show that the range of communication delay over which it is better to have a percentage of distributed transactions increase as the data access time increases. But for a given data access time and an arrival rate of transactions, it

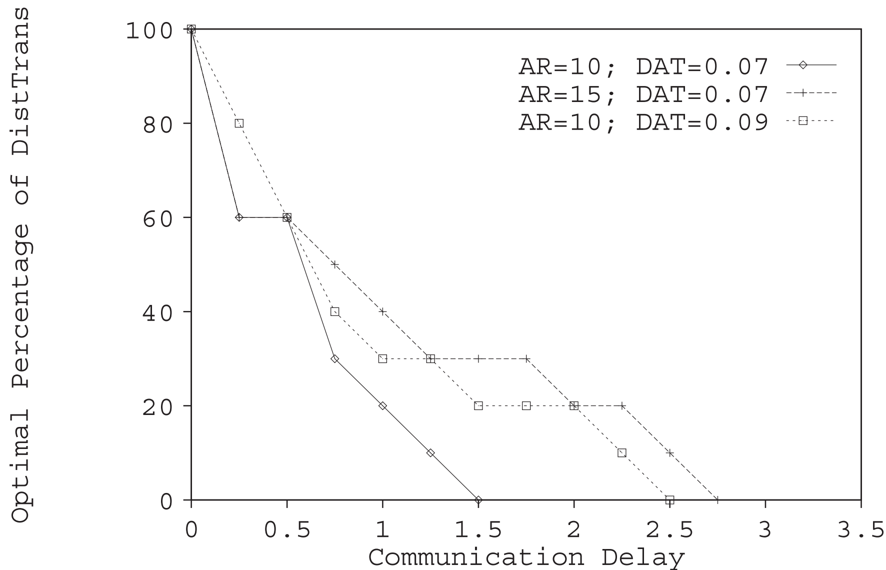


Figure 34: Optimal Distr Trans Percentage Over CommDelay

will be of interest to know what these optimal distributed transaction percentages are over this range of communication delay. The simulation results shown in Figure 34 illustrate the optimal percentage of distributed transactions for three combinations of arrival rates and data access time. The first two cases are when the data access time is 0.07 seconds but the arrival rates are 10 transactions/sec and 15 transactions/sec. In both these cases for zero communication delay it is better to have all distributed transactions (100%). But as soon as communication delay increases to 0.25 seconds, it is better to have only 60% of transactions as distributed. This is because local transactions can access the data when distributed sub transactions are waiting because of the communication delay. This increases the over all concurrency in executing the transactions, and thus decreases the average transaction response time. For the arrival rate 10 transactions/sec the optimal percentage of distributed transactions quickly drops to zero when the communication delay is 1.5 seconds. But when the arrival rate is 15 transactions/sec there is more data contention and the optimal distributed transactions percentage drops slowly to zero for the communication delay of 2.75 seconds. An increase in data access time with same arrival rate has the same effect as that of increasing the arrival rate. That is, it increases the data contention, which means a slow drop in optimal distribution transaction percentage as communication delay

increases. With the arrival rate of 10 transactions/sec and data access time as 0.09 seconds, the optimal distributed transaction percentage drops slowly to zero for the communication delay of 2.5 seconds.

There is a range of communication delay where the optimal distributed transactions percentage remains fairly steady (especially when there is a slow drop in optimal distributed transactions percentage as communication delay increases). This range is [1.5:2.0] seconds with optimal distributed transaction percentage being 20, when arrival rate is 10 transactions/sec and data access time is 0.09 seconds, and it is [1.75:2.25] seconds with optimal distributed transaction percentage being 30, when arrival rate is 15 transactions/sec and data access time is 0.07 seconds. Having such ranges of communication delays with a steady optimal distributed transactions percentage is helpful in selecting the network environment for the distributed database system. This also gives more freedom in allocating the fragments. As increasing the communication delay does not affect the optimal percentage of distributed transactions. But for the cases when there is a steep drop in optimal distributed transactions response time, there is very little flexibility in choosing the network configuration or little freedom in allocating the data.

7.7.3.1 Practical Applicability

In the previous subsection, the tradeoff between the data access time and communication delay was presented. But there was no connection between the parameter values used and the applications being executed in reality. In this section the parameter values are used to specify the amount of data that can be accessed from different sites based on the network characteristics. A distributed transaction spends twice the time for communication delay on the data transfer. Hence if the communication delay is 1 second, then 2 seconds of the response time are used for data transfer. Moreover, the data can be transferred concurrently between many sites.

Table 7 shows the maximum amount of data that can be transferred between two sites in the distributed transaction processing environment for various network data transfer rates. The network data transfer rates considered are 128Kbs, 1.544Mbs (T1 line), 15Mbs (T3 line), 45Mbs, 150Mbs (FDDI), and 625Mbs. In future it should be possible to lease lines that can transfer data at the rates of gigabits per second. In distributed relational database systems, the distributed query processor optimizes the transactions so that only the necessary data is transferred between the sites. This is done by using local query optimization

Table 7: Maximum Data Transfer for Distributed Transaction Processing

Communication Delay	Data Transfer Rates 128 Kbps	1.544 Mbps	15 Mbps	45 Mbps	150 Mbps	625 Mbps
0.25 Sec	8K	96.5K	937.5K	2.8M	9.375M	39M
0.5 Sec	16K	193K	1.875M	5.6M	16.8M	78M
1.0 Sec	32K	386K	3.75M	11.2M	33.6M	156M
2.0 Sec	64K	772K	7.5M	22.4M	67.2M	312M
2.5 Sec	80K	965K	9.375M	28M	84M	390M

and semi-joins. Figure 32 shows that it is beneficial to distribute all the transactions when the communication delay is less than 0.25 seconds. Therefore maximum amount of data that can be transferred between a coordinating site and each of the subtransaction sites varies from 8Kbytes to 39Mbytes, for data transfer rates of 128Kbps to 625Mbps. This is an extremely wide range. Therefore, at higher data transfer rates (45Mbps or greater) there will still be some application processing environments that fall within this range. At lower data transfer rates it is difficult to support distributed sub transactions that access large amounts of data and send it to the coordinating site.

Figure 32 shows that the communication delay range within which it is beneficial to have some distributed transactions increases as the data access time increases. For some data access times we now show the maximum amount of data that can be transferred by the distributed transactions. When the data access time is 70 milliseconds, the range of communication delay over which it is better to have some percentage of distributed transactions is [0.0–1.0] seconds. The maximum data that can be transferred with communication delay of 1 second varies from 32Kbytes to 156Mbytes for data transfer rates from 128Kbps to 625Mbps. Note that even for smaller data transfer rates the maximum amount of data that can be transferred between each of the subtransactions and the coordinator of the distributed transaction is 32Kbytes. Therefore, at higher data transfer rates (15Mps or greater) there will be application processing scenarios that fall within this range. This is more evident when the data access time increases to 90 milliseconds, and the communication delay range over which it is better to have some percentage of distributed transaction is [0.0–2.0] seconds. The maximum data that can be transferred with communication delay as 2 seconds varies from 64Kbytes to 312Mbytes for data transfer rates from 128Kbps to

625Mbps. Therefore, in this case also there will be application processing scenarios that transfer this amount of data to process the distributed transactions. Similarly, it can shown for the case when the arrival rate of transactions is 15/second that there will exist applications that do have data transfer within the specified maximum limits to process the distributed transactions.

From the above discussion it can be seen that this particular classification of a range of communication delay into three parts, namely, all distributed transactions, some percentage of distributed transactions, and all local transactions is meaningful, and can be used by the distributed database designer in selecting the right distributed database design for a class of applications. These ranges of communication delays over which it is better to have some percentage of distributed transactions will vary as the parameters of the distributed database environment change. Depending upon the communication technology being used, one would select an appropriate communication delay in the computation.

7.7.4 Effect of Number of Data Accesses

In this experiment data access time was fixed at 50 milliseconds and the communication delay was fixed at 1 second. For a fixed number of data accesses for each sub transaction of a distributed transaction, the number of data accesses for local transaction was changed and vice versa.

In Figure 35, the number of data accesses for a local sub transaction of a distributed transaction was fixed at 10 per distributed sub transaction, and the number of data accesses was varied from 20 to 40 for local transactions. The transaction arrival rate was fixed at 15 transactions per second. When the number of data accesses for local transactions was 20 or 30, there were enough data accesses to take advantage of the waiting time of the distributed transactions due to the communication delay. Also the number of data accesses for local transactions was small enough not to have high lock contention. Therefore the least average transaction response time is when all transactions are being executed are local. But when the number data accesses for local transactions is increased to 40, there is too of contention for locks. This increases the transaction response time for local transactions. Therefore, the average transaction response time is increased. The least average transaction response time is got for 50 percent of local transactions. This is because by decreasing the percentage of local transactions, the lock contention is decreased, and the processing is distributed.

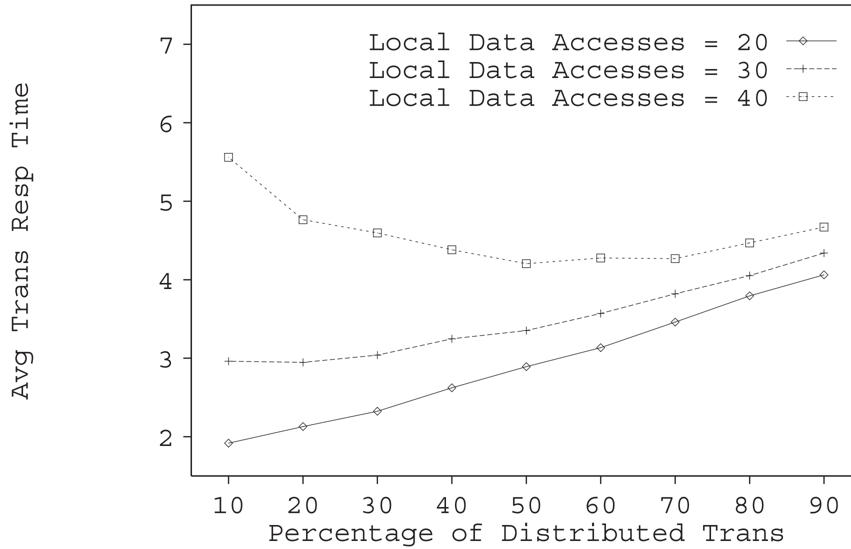


Figure 35: Effect of Change in Number of Local Data Accesses

Therefore, if the number of local data accesses is large then it is not beneficial to have all local transactions.

Figure 36 shows the effect of change in number of data accesses for the sub transaction of a distributed transaction, while keeping the number of data accesses for a local transaction constant. The number of data accesses for local transactions was kept fixed at 30 per transaction, while the number data accesses for a sub transaction of a distributed transaction was varied from 5 to 15. At small values number of data accesses for local sub transaction of a distributed transaction it is beneficial to have all local transactions executing to get least average transaction response time, but as the number of data accesses for the local sub transaction of a distributed transaction increase, the lock contention between local transaction and the local sub transaction of the distributed transaction increases, which increases the average response time of local transactions. Note that when a local transaction has lock contention with a sub transaction of a distributed transaction, the local transaction has to wait till the distributed transaction commits or aborts. From the Figure 36 it is seen that 80% of local transactions give the least average transaction response time. This is

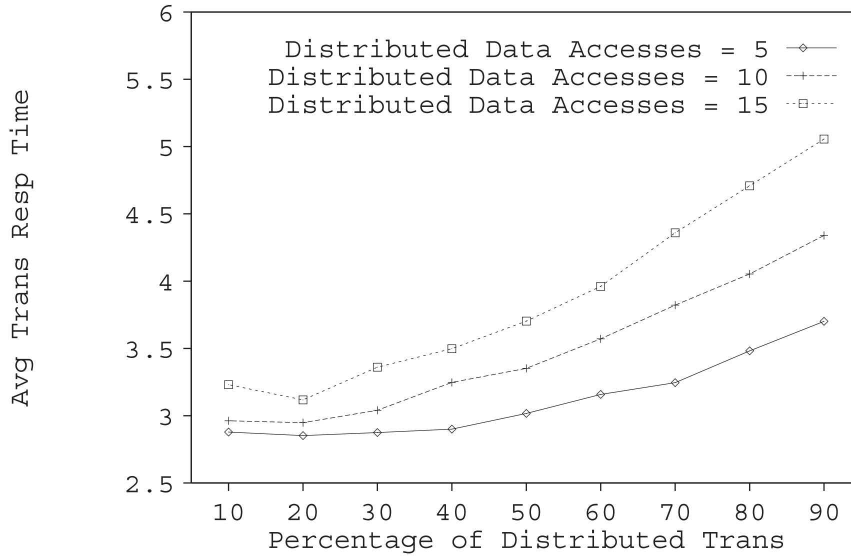


Figure 36: Effect of Change in Number of Distributed Data Accesses

because reducing the percentage of local transactions reduces the contention with sub-transactions of the distributed transactions. Thus if the number of data accesses for a sub transaction of a distributed transaction is large then it is beneficial to reduce the percentage of local transactions so as to get least average transaction response time.

7.8 Summary

In this chapter, an event-based simulation model to estimate the average transaction response time in a distributed database environment was presented. This simulation model is based on a model of locking mechanism which has been widely accepted for modeling data contention between transactions. The data contention has been modeled analytically by calculating the probability of data contention using an analytical formulae. The cyclic interaction between the probability of contention and the transaction response time has been captured in the simulation model. The effect of changes in data access time, communication delay, arrival rate of transactions, and number of data accesses has been presented. A comprehensive set of simulations were done to generate optimal percentage of distributed transactions for a range of data access time and communication delay. It was shown that

for a large range of communication delay and data access time it is beneficial to have some percentage of distributed transactions. Thus the tradeoff between amount of time spent on data access and communication delay has been studied. The effect of percentage of local transactions on the lock contention was presented, with the conclusion that if the number of data accesses for local transactions or sub transactions of distributed transactions is large then it is beneficial to reduce the percentage of local transactions. This is the first simulation study that was done by considering a range of changes like communication delay, data access time, arrival rate, and number of data accesses for a distributed database environment.

CHAPTER 8

A DISTRIBUTED DATABASE REDESIGN METHODOLOGY

8.1 General Comments

In the previous chapters, a methodology for the design of a distributed database, algorithms to efficiently materialize redesigned distributed databases, and a simulation model to measure the average transaction response time for an application executing on a given distributed database design have been developed. In this chapter an integrated methodology for distributed database redesign process in an *application processing center* environment is presented. In such an environment a set of well defined applications are executed over time. Some applications are executed all the time *(perpetual applications)* and others are executed during some intervals of time *(non-perpetual applications)*. Therefore, it is these non-perpetual applications that motivate the need for a redesign methodology. This is because no single distributed database design can be optimal for all applications when some non-perpetual applications are executed over different intervals of time. It is necessary to consider a possible change in design whenever a non-perpetual application is initiated for execution. This is done to ensure efficient execution for all the applications in the distributed database environment.

In this chapter, a model for application processing center scenario is presented, the viability of this approach is explored, and finally a small example illustrating the technique used is presented.

8.2 A Model for the Application Processing Center Scenario

Given A_1, A_2, ..., A_n, where each A_i consists of a class of applications and each application class is defined by a set of transactions initiated by all the applications in that class. A transaction accesses one or more relations of the distributed database. These classes of applications have included in them all the perpetual applications. The main characteristic of this application processing center scenario is that at some pre-specified time points (known

as *change points*) a class of applications is initiated and the time duration of its execution known. Given that a class of applications is being executed currently, then the new class of applications to be executed at the next change point can be one of the classes of applications. At each change point there is a probability of selecting a class of applications to be initiated. Each class of applications is executed on a single distributed database design. There are a set of candidate distributed database designs from which one is selected for a class of applications to run on. This dynamic process of selecting, initiating, and executing the class of applications at the change points on a distributed database design can be modeled as a *discrete Markov process.*

A stochastic matrix $P = [p_{ij}]$ gives the probability of changing from executing class of applications A_i before a change point to executing class of applications A_j after the change point, where $1 \leq i, j \leq n$. After each change point, a new class of applications can be executed on any one of the alternative candidate distributed database designs. Each alternative distributed database design has a cost value for it to be selected. This cost corresponds to the cost of executing the class of applications on the distributed database design plus the cost of materializing the new design. The objective is to minimize the cost of processing all the applications in the distributed database environment.

Therefore, the problem to efficiently process all the classes of applications in an application processing center environment can be specified as:

"Given a series of change points and the probabilities of change in processing classes of applications over the time period in an application processing center scenario, generate the optimal series of distributed database designs that need to be used for the duration of the time interval between successive change points".

This problem can be solved by using the *Markovian decision analysis* proposed by Howard [How60] in his thesis. The states of the Markov process are $s_1, s_2, s_3, \ldots, s_{n \times d}$, where d is the number of distributed database designs on which the classes of applications can be executed. Each state represents a class of applications executing on a distributed database design and is specified by a pair identifying the class of applications and the distributed database design on which it is executed. As there are n classes of applications and d designs, the number of states in the discrete Markov process is $N = n \times d$. Note that a class of applications can be executed on only one out of the d possible distributed database designs. Also note that there are n possible classes of applications that can be executed after a change point. That is, after a change point, for each of the n possible classes of

applications to be executed next, we can evaluate the cost of using each of the d distributed database designs.

Given that the current state of the application processing center scenario has class of applications A_i executing on design D_j, then the following alternative state transitions can occur. Each alternative is described by the following redesign policy:

Table 8: A Redesign Policy

	Application A_1	Then use Design D_{i_1}
	Application A_2	Then use Design D_{i_2}
	Application A_3	Then use Design D_{i_3}
If the Application A_i changes to	.	.
	.	.
	.	.
	Application A_n	Then use Design D_{i_n}

Note that each of the D_{i_k} where $1 \leq i_k \leq n$ range over all possible distributed database designs. Hence there will be d^n possible alternatives. The problem then can be restated as that of selecting the best possible policy among the d^n policies for each state of the discrete Markov process to efficiently process all classes of the applications. In order to do, this a column policy vector O with N elements, each of whose element is a particular policy o for each state of the Markov process is defined. The algorithm to generate the optimal redesign policy starts with an initial redesign policy vector and uses Howard's value iteration and policy iteration algorithms to iteratively generate the optimal policy vector. This optimal policy vector is used by the distributed database administrators to select the distributed database design to be used after each change point.

Figure 37 shows the discrete Markov process representing the processing of classes of applications in the application processing center scenario. At the first change point class of applications, A_i is executed on distributed database design D_r, at the next change point class of applications, A_j is executed on distributed database design D_s, and so on. Note that in this Markov process there is no change in distributed database design being used after the third change point, and same class of applications are executed after the fifth change point but on a different distributed database design. Also shown in Figure 37 is a redesign policy for the state defined by (A_i, D_j). This redesign policy states, that if after the change point, that class of applications A_1 is to be executed, then use distributed database design

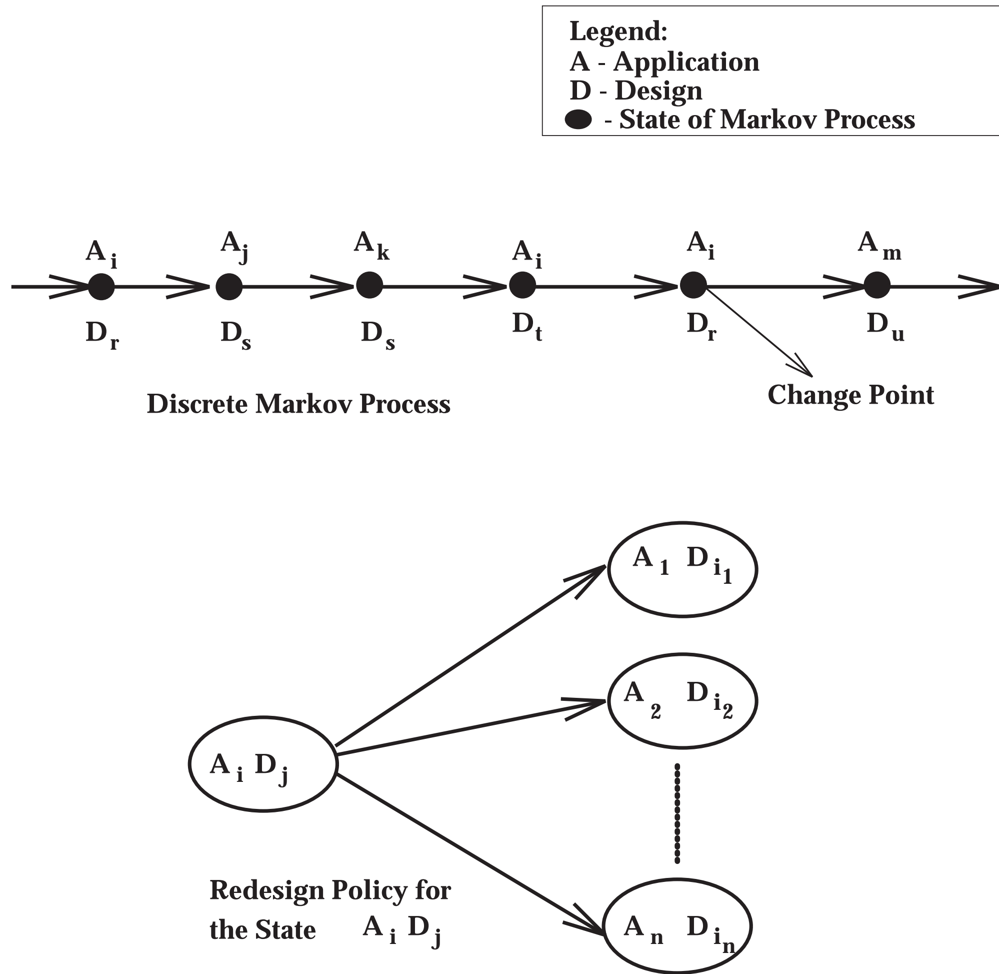


Figure 37: Discrete Markov Process and Redesign Policy

D_{i_1}, else if A_2 is to be executed, then use distributed database design D_{i_2}, ..., and, if A_n is to be executed, then use distributed database design D_{i_n}.

The detailed algorithms to generate the optimal policy vector are given in [How60] and are not presented here. In order to use this technique in practical business data processing environment, following aspects of the problem need to addressed:

- How are the classes of applications defined?
- How are the change points and execution durations generated?
- How is the stochastic matrix P generated?
- How are the candidate distributed database designs generated?

- How are the cost values to generate the optimal policy vector calculated?
- Finally, how is the optimal redesign policy generated?

8.3 Application Classes

In any business data processing environment using an application processing center concept based on a distributed database environment, an application consists of any of the following:

- A large integrated module of applications that support wide variety of functions. These applications are typically on-line transaction processing applications based on a set of menus. In case of such large applications either the complete application with all its sub-modules is executed as a class of applications. Or, if there are large number of modules in the application then the modules can be grouped according to their frequency of execution into different classes of applications. A report generating module which is executed once a week can can be treated as a class of applications with only one application. The best way to classify such large applications is based on whether the application is executed as a whole or whether it is broken up into classes of applications each of which is executed as a whole application. Bank-teller machine application consists of a sieve of applications which are executed as a single large application. But the admission information system of an university, consists of a set of applications which can be redefined as classes of applications like "registration package", "grade reporting package", "class scheduling package", etc., such that each package is executed as a whole at some pre-specified time points of the year.
- The next kind of applications are the large batch applications which are executed at some pre-specified times. These batch jobs process large volumes of data and generate reports, or update the database relations. All business data processing centers run batch jobs periodically, typical examples are payroll processing, insurance claim processing, daily bank account updates, etc. Just like large on-line applications, the batch applications are well defined and well tested applications. Batch applications take up lot of resources and usually take long times to execute. Hence it is important to efficiently execute the batch jobs. A single batch job initiates large number of transactions during its execution time. These transactions are well defined and are characterized into a number of transaction types. Each transaction type of the

batch application defines an application of the class of applications. If more than one batch job is initiated at the same time, then they can be grouped to form a class of applications.

- There are small scale applications which are developed based on some business need, and are executed over some periods of time. If these kind of applications access large amount of resources in the distributed database environment or if their performance is very crucial then they can be defined as a class of applications. Also, ad hoc queries or transactions which are executed periodically or at some pre-specified time points can also be grouped as class of applications.

A well defined set of transactions in the distributed database environment which takes up a large amount of resources, and needs to be executed efficiently at different intervals of time can be grouped as a class of applications for the purposes of the above redesign methodology. Most database management systems supply utilities to collect and report data regarding resource consumption by the transactions, frequency of their initiation, the relations they access, etc. This information can be used by the distributed database administrators to define the above classes of applications.

8.3.1 Change points and Execution Duration

In any production business data processing environments, a set of well defined applications are executed according to a well defined schedule. This schedule is governed by the requirements for processing of the data by various business policies. There are some periods of time when some classes of applications are executed, and other periods of time when they are not executed. Each initiation of the application is treated as a change point because the distributed database system has to efficiently process the new set of transactions in addition to those it is already processing. Similarly, termination of the execution of an application is considered as a change point. The duration between the application initiation and application termination is the execution duration for the application. This schedule gives the time points when the applications are initiated. Since not all applications are executed all the time, there is a duration for each initiation of the applications execution time. The duration for each initiation of the application can be tabulated and averages calculated. The distributed database administrators can collect this information by monitoring the processing of the applications.

Large on-line or batch applications tend to have longer execution durations than small applications or transactions. Hence the distributed database administrators have to evaluate all the applications based on execution duration, resources consumed, and importance to business processing. This evaluation helps in short listing and defining classes of applications, change points and execution durations for these applications. As pointed out earlier, most of this information can be collected by monitoring the applications being processed and from the statistics collected by the database management systems.

8.3.2 Calculating the Probability of Change

The dynamic process of initiating and executing the applications in the application processing center scenario is modeled as a discrete Markov process. In order to define this Markov process, the probability of change in the class of applications to be executed from "before the change point" to "after the change point" needs to be calculated. The stochastic matrix P gives the probabilities of changing from processing one application to processing any other application after a change point. The previous section shows how the distributed database administrators define a set of classes of applications which need to be efficiently executed. Following are some of the ways in which the distributed database administrators can calculate the probabilities in the stochastic matrix.

Uniformity: Treat all the classes of applications to be of the same importance, and any application can be initiated at each check point. Then $p_{ij} = 1/n$ for all, $1 \leq i, j \leq n$. This case will arise when there are too many applications and applications are initiated and executed randomly.

Schedule: In this case, the initiation of classes of applications follows a schedule strictly and it is known exactly which class of application is initiated after a change point. If after the termination of a class of application A_i, the class of applications A_j is always initiated, then $p_{ij} = 1$.

Empirical: The relative frequencies of classes of applications initiated after the termination of a class of application is used to calculate the probability of change. This is an empirical approach because it is based on monitoring the application processing and collecting the statistics. The processing of these classes of applications is monitored and the frequency of executing each class of applications is calculated.

In an application processing center scenario, any one of the above methods or a combination of the methods based on the statistics collected can be used.

8.4 Generation of Candidate Distributed Database Designs

In order to apply the technique of Markov decision analysis to generate the optimal policy vector, a set of candidate optimal distributed database designs based on application processing characteristics need to be generated. The classes of applications are well defined. Therefore, the transactions initiated, the frequencies with which they are initiated, the columns accessed, the predicates used to access the tuples of the relations, the average length of a tuple of a relation, and the cardinality of the relation are known in advance. In chapters 3 and 4 a mixed fragmentation methodology was developed. This methodology generates a fragmentation and allocation scheme based on the characteristics of the transactions and the sites from which they are initiated. Each class of applications initiates a set of transactions, each with some frequency from a site. This information, along with the information about columns and tuples of the relations accessed by each of the transactions is sufficient to design a distributed database. Each class of applications imposes different constraints on the distributed database design; therefore no single distributed database design can efficiently process all classes of applications. That is, it is very much possible that the optimal distributed database design for all classes of applications is outperformed by an optimal design for a class of applications.

The mixed fragmentation methodology generates the optimal distributed database design for each class of applications. This optimal design efficiently processes the class of applications, but may give a very bad performance for a different class of applications, thus requiring a redesign. The materialization of redesign is costly and frequent redesign of the distributed database is not cost effective. Therefore, a sub-optimal design which processes both the classes of applications with acceptable performance and eliminates frequent redesign will be cost effective in the long run. Thus, in addition to the optimal designs for each of the classes of application additional optimal designs for some subsets of classes of applications need to be generated. The stochastic matrix P giving the probabilities of change in the class of applications can be used to decide which subsets of classes of applications need to be considered for generating the candidate distributed database designs. The three techniques to calculate the probabilities ("uniformity", "schedule", and "empirical") described earlier need to considered to generate the candidate distributed database designs.
Uniformity: In this case every class of applications has same probability of being initiated at a change point. Therefore, in addition to optimal distributed database designs

for each class of applications, we can consider optimal distributed database designs for a combination of each subset of classes of applications. Hence if there are n classes of applications, then the number of candidate distributed database designs is 2^n. The number of candidate distributed database designs possible grows exponentially with the number of classes of applications. Among all these possible candidate distributed database designs a maximum of $n \times N$ designs will be used. This is because each policy vector can use maximum of N different distributed database designs and there are N policy vectors. Therefore, once the optimal policy vector is generated, then all but maximum of $n \times N$ candidate distributed database designs can be eliminated. But generating all possible candidate distributed database designs will allow the Markov decision analysis to select those which process all the classes of the applications efficiently in the long run. In some cases, it is clear from the application processing scenario that some combinations of the classes of applications need not considered to generate candidate distributed database design.

Schedule: The schedule defines which class of applications is initiated after a change point. Hence there is a series of different classes of applications being initiated and executed over successive change points. Because of the execution durations between consecutive change points, the class of applications being executed are related to the next class of applications executed most than any other class of applications. Two or three consecutive classes of applications executed are combined to generate the candidate distributed database designs. The generation of the candidate distributed database designs is more focused than the *equality case* and many of the 2^n possible candidate distributed database designs are eliminated.

Empirical: The priority case specifies which subset of classes of applications that can be initiated after a change point. The classes of applications initiated over number of consecutive change points forms a tree rooted at the class of applications first initiated. The successive two or three classes of applications initiated from each possible branch of the tree generate the possible candidate distributed database designs. Again, this approach eliminates right away large number of 2^n possible candidate distributed database designs. Note that this case falls between the above two cases.

In general, the number of candidate distributed database designs depends on the variation in the data accessed by the classes of applications. If there is a large amount of variation, then a distributed database design for one class of applications will provide bad performance compared to another class of applications. But if there is a very little variation

then a single distributed database design will provide reasonable performance to most of the classes of applications. We will illustrate this with an actual case study in Chapter 9.

The above set of approaches based on the characteristics of the discrete Markov process generate the set of possible candidate distributed database designs. These set of candidate distributed database designs are evaluated while generating the optimal policy vectors by using the Markovian decision analysis. The distributed database designs that are most efficient to maintain, reorganize and efficiently process all the applications are used and the rest are eliminated.

8.5 Calculating the Cost Values

There are two types of cost values that need to be calculated for being used in the algorithms to generate the optimal policy vectors. The first type of cost value is the cost of materializing redesigned distributed databases. The second type of cost value is the cost of using a candidate distributed database design for processing a given class of applications. These cost values are calculated by using the algorithms and techniques developed in chapters 5 and 7. A synopsis of these approaches is given below.

Materialization Cost: Given the lengths of the columns of each of the relations, and the cardinalities of each of the horizontal fragments the time taken to materialize the new design is given by equation 5. This time taken to materialize the populated distributed database is the cost for materialization. The equation 5 takes into consideration only the data transfer and data access costs but not the CPU costs and queuing delay cost. These costs are assumed to be negligible in comparison to the data transfer and data access costs. Let C_M denote the cost of materialization.

Transaction Processing Cost: The class of applications are executed on a distributed database design based on a distributed database environment. In chapter 7 a simulation model is presented to estimate the average transaction response time based on the frequency of transactions initiated, the number of transactions initiated, the fragmentation and allocation schemes, and the system parameters (like data access time, communication delay and MIPS of the CPU). The average number of transactions initiated by a class of applications and their frequencies is calculated by monitoring the applications processed and collecting the statistics. Since the applications to be processed and the application processing center are well defined, it is easy to develop utilities to collect these statistics.

The database management system provides tools that calculate the number of transactions initiated from a given site and the times when these transactions were initiated. Therefore based on the average transaction response time r and average number of transactions l initiated, the transaction processing cost t in seconds is given as:

$$t = \frac{C_M}{l} + r \tag{11}$$

Note that the simulation model used to estimate the average transaction response time takes into consideration the effect of concurrently executing the transactions.

The objective is to minimize the total time taken to execute all the classes of applications in an application processing center based on a distributed database environment. This total time takes into consideration the materialization of redesign cost whenever there is a change in redesign and the time taken to process all the transactions initiated by the applications. An approach to achieving this is by using the Markov decision analysis.

8.6 Optimal Policy Vector Generation

The optimal policy vector generation is based on the Markovian decision analysis technique developed by Howard [How60] in his PhD thesis. The algorithms for generating these optimal policy vectors are well known. Here a brief description of the technique along with an example is presented. A more detailed example going through all phases of the distributed database redesign methodology is presented in the next chapter. In this section we shall show how the redesign problem can be modeled as a Markovian decision analysis problem.

8.6.1 Modeling of Redesign Problem

Given n classes of applications $A_1, A_2, \ldots, A_n$ that need to be processed in the application processing center environment. These applications can be executed on any one of the d candidate distributed database designs. Let $P = [p_{ij}]$ be the stochastic matrix giving the probability of executing class of applications A_j after completing the execution of class of applications A_i. $C = [c_{ij}]$ be the cost matrix giving the time taken to materialize design d_j from the d_i. $R = [r_{ij}]$ is the matrix giving the time taken to execute the class of applications A_i on design d_j. Note that P is a matrix of order $n \times n$, C is a matrix of order $d \times d$, and R is a matrix of order $n \times d$.

The state of the discrete Markov process is the pair defined by (application, design used by the application). Therefore, there are $n \times d$ states in the discrete Markov process. The state transition probabilities are defined by the policy vector being used by the application processing center. A policy vector is a column vector with $n \times d$ elements, one for each state of the Markov process. Each element of the policy vector defines the policy that needs to be used. The policy gives which design needs to be used after a change point from the current state.

Let us consider an example with 3 applications and 3 distributed database designs. Let the stochastic matrix P giving the probability of change

$$P = \begin{bmatrix} 0.8 & 0.0 & 0.2 \\ 0.2 & 0.8 & 0.0 \\ 0.0 & 0.2 & 0.8 \end{bmatrix}$$

If the current class of applications being executed is A_1, then after the change point, with a probability of 0.8 the class of applications A_1 continue to be executed, and with probability 0.2 the class of application A_3 will be executed. Similarly the class of applications A_2 continue to be executed with probability 0.8, and change with probability 0.2 to the class of applications A_1. And the class of applications A_3 continue to be executed with probability 0.8, and change with probability 0.2 to the class of applications A_2.

The matrix C gives the cost of changing from one distributed database design to another.

$$C = \begin{bmatrix} 0.0 & -15.0 & -10.0 \\ -10.0 & 0.0 & -15.0 \\ -15.0 & -10.0 & 0.0 \end{bmatrix}$$

If there is no change in distributed database design to be used after a change point then it has zero cost. The cost materializing distributed database design D_2 from D_1, D_3 from D_2, and D_1 from D_3 are all 15 seconds[1]. And the cost of materializing distributed database design D_3 from D_1, D_1 from D_2, and D_2 from D_3 are all 10 seconds.

The matrix R gives the cost of executing a class of applications on a distributed database design.

[1]Since the Markov Decision Analysis maximizes benefit values, and the goal of redesign policy is to minimize cost, we represent cost values as negative benefit

$$R = \begin{bmatrix} -10.0 & -15.0 & -15.0 \\ -15.0 & -10.0 & -15.0 \\ -15.0 & -15.0 & -10.0 \end{bmatrix}$$

The cost of executing class of applications A_1 on distributed database design D_1 is the cheapest at 10 seconds, and on other designs it takes 15 seconds. Similarly, the costs of executing A_2 on D_2, and A_3 on D_3 is 10 seconds, and on all other distributed database designs it is 15 seconds. For these three classes of applications, the following observations can be made:

- The probability of a class of application to continue executing after a change point is four times the probability that there will be a change in class of applications to be executed. This implies that on an average a class of applications can be expected to be executed over four change points. Moreover, there is a well defined schedule for the change, which is $A_1 \longrightarrow A_3 \longrightarrow A_2 \longrightarrow A_1$.

- Each distributed database design executes a single class of applications optimally. The class of applications A_1 are efficiently executed by distributed database design D_1, A_2 by D_2, and A_3 by D_3. Therefore, it is preferred to execute a class of applications on the distributed database design that best executes it.

- The materialization of redesign cost is the cheapest along the schedule. That is, cost of materializing D_3 from D_1 is cheaper than D_2 from D_1, cost of materializing D_1 from D_2 is cheaper than D_3 from D_2, cost of materializing D_2 from D_3 is cheaper than D_1 from D_3. Therefore, these costs encourage the redesign in order to benefit the schedule of the execution of classes of applications. Also the cost of any change in distributed database design just prior to executing a class of applications is amortized over four change points on an average.

The cost values used in the example are taken so as to emphasize the technique being used. In chapter 9 the example will use cost values based on the case study undertaken. In the next section a formalization of the technique will be presented.

The Markovian decision analysis process consists of two phases *value-determination process* and the *policy improvement process*. The process starts by considering an initial

Table 9: Redesign Policies for 3 Applications and 3 Designs

Policy Number	Policy
1	$(A_1 \to D_1), (A_2 \to D_1), (A_3 \to D_1)$
2	$(A_1 \to D_1), (A_2 \to D_1), (A_3 \to D_2)$
3	$(A_1 \to D_1), (A_2 \to D_1), (A_3 \to D_3)$
4	$(A_1 \to D_1), (A_2 \to D_2), (A_3 \to D_1)$
5	$(A_1 \to D_1), (A_2 \to D_2), (A_3 \to D_2)$
6	$(A_1 \to D_1), (A_2 \to D_2), (A_3 \to D_3)$
7	$(A_1 \to D_1), (A_2 \to D_3), (A_3 \to D_1)$
8	$(A_1 \to D_1), (A_2 \to D_3), (A_3 \to D_2)$
9	$(A_1 \to D_1), (A_2 \to D_3), (A_3 \to D_3)$
10	$(A_1 \to D_2), (A_2 \to D_1), (A_3 \to D_1)$
11	$(A_1 \to D_2), (A_2 \to D_1), (A_3 \to D_2)$
12	$(A_1 \to D_2), (A_2 \to D_1), (A_3 \to D_3)$
13	$(A_1 \to D_2), (A_2 \to D_2), (A_3 \to D_1)$
14	$(A_1 \to D_2), (A_2 \to D_2), (A_3 \to D_2)$
15	$(A_1 \to D_2), (A_2 \to D_2), (A_3 \to D_3)$
16	$(A_1 \to D_2), (A_2 \to D_3), (A_3 \to D_1)$
17	$(A_1 \to D_2), (A_2 \to D_3), (A_3 \to D_2)$
18	$(A_1 \to D_2), (A_2 \to D_3), (A_3 \to D_3)$
19	$(A_1 \to D_3), (A_2 \to D_1), (A_3 \to D_1)$
20	$(A_1 \to D_3), (A_2 \to D_1), (A_3 \to D_2)$
21	$(A_1 \to D_3), (A_2 \to D_1), (A_3 \to D_3)$
22	$(A_1 \to D_3), (A_2 \to D_2), (A_3 \to D_1)$
23	$(A_1 \to D_3), (A_2 \to D_2), (A_3 \to D_2)$
24	$(A_1 \to D_3), (A_2 \to D_2), (A_3 \to D_3)$
25	$(A_1 \to D_3), (A_2 \to D_3), (A_3 \to D_1)$
26	$(A_1 \to D_3), (A_2 \to D_3), (A_3 \to D_2)$
27	$(A_1 \to D_3), (A_2 \to D_3), (A_3 \to D_3)$

policy vector, determining the values, and evaluating if the policy could be improved further. This iteration between value-determination and policy improvement ends when in successive iterations there is no change in the policy vector.

A policy vector defines the Markov decision process that needs to be used by the application processing center to process all its classes of applications. In case of the example defined in the previous section, we have 9 states and 27 policies. The policy vector defines policy adopted by each of the states. Table 9 lists all the policies for this example. Each

policy, say for example, $(A_1 \rightarrow D_2), (A_2 \rightarrow D_1), (A_3 \rightarrow D_3)$ (i.e. policy number 12) means that if the new class of applications to be executed is A_1 then use design D_2; if it is A_2 then use design D_1; if it is A_3 then use design D_3.

Howard's algorithms use the input provided by the matrices P, C, and R, to generate the optimal policy vector after five iterations. This process of optimal policy vector generation can be described as follows:

$It\#0$	$It\#1$	$It\#2$	$It\#3$	$It\#4$	$It\#5$
	$g = -13.33$	$g = -13.33$	$g = -13.33$	$g = -12.0$	$g = -12.0$
$1 \rightarrow$	$1 \rightarrow$	$1 \rightarrow$	$3 \rightarrow$	$3 \rightarrow$	3
$1 \rightarrow$	$1 \rightarrow$	$2 \rightarrow$	$2 \rightarrow$	$2 \rightarrow$	2
$1 \rightarrow$	$3 \rightarrow$	$3 \rightarrow$	$21 \rightarrow$	$3 \rightarrow$	3
$1 \rightarrow$	$1 \rightarrow$	$4 \rightarrow$	$4 \rightarrow$	$4 \rightarrow$	4
$1 \rightarrow$	$4 \rightarrow$	$4 \rightarrow$	$4 \rightarrow$	$4 \rightarrow$	4
$1 \rightarrow$	$1 \rightarrow$	$4 \rightarrow$	$21 \rightarrow$	$4 \rightarrow$	4
$1 \rightarrow$	$1 \rightarrow$	$1 \rightarrow$	$3 \rightarrow$	$3 \rightarrow$	3
$1 \rightarrow$	$4 \rightarrow$	$5 \rightarrow$	$5 \rightarrow$	$5 \rightarrow$	5
$1 \rightarrow$	$3 \rightarrow$	$6 \rightarrow$	$6 \rightarrow$	$6 \rightarrow$	6

The It# denotes the iteration number and g denotes the gain value. The iteration converges to an optimal policy vector with a gain value of -12. The optimal policy vector is:

$$\mathcal{P} = \begin{bmatrix} 3 \\ 2 \\ 3 \\ 4 \\ 4 \\ 4 \\ 3 \\ 5 \\ 6 \end{bmatrix}$$

The discrete Markov process defined by this policy vector is illustrated in Figure 38. This Markov process swiftly converges to the process defined by $(A_1, D_1) \rightarrow (A_2, D_2) \rightarrow$

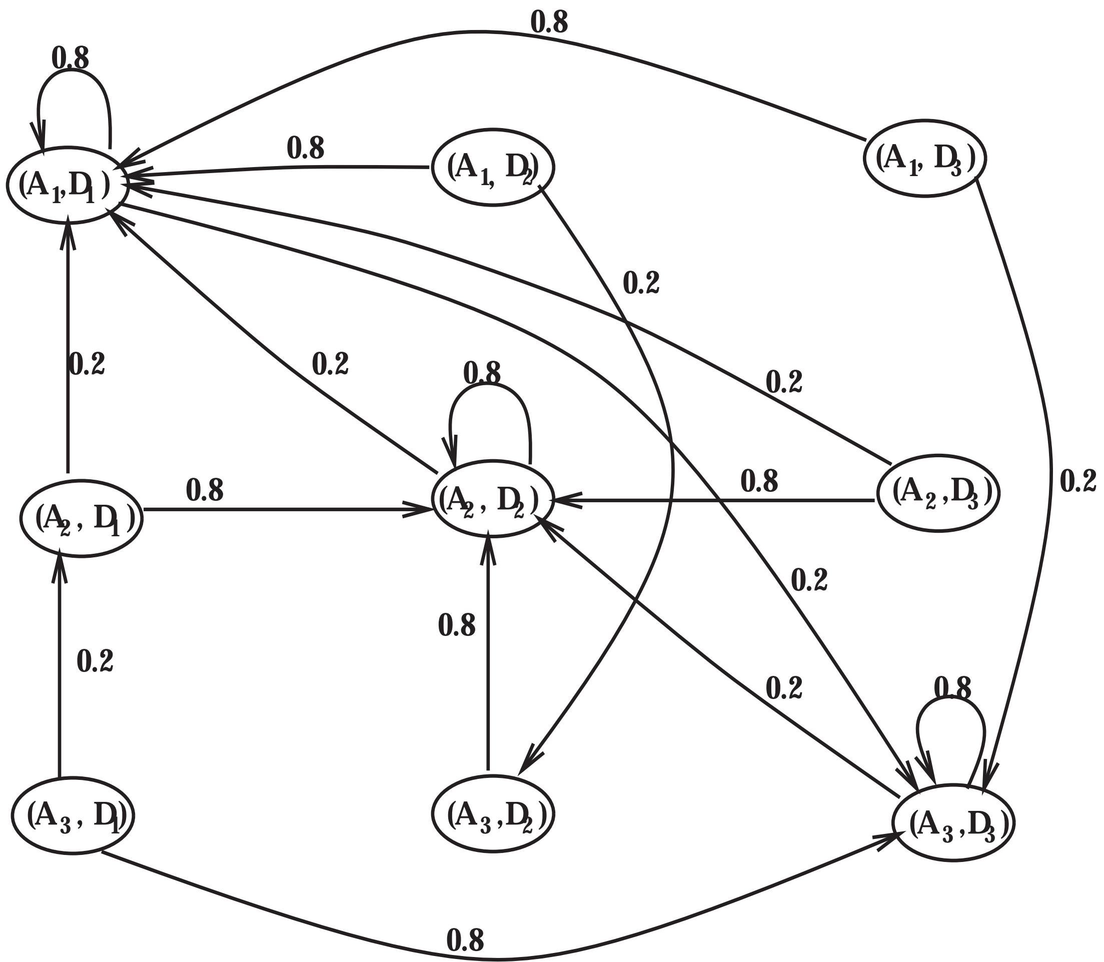


Figure 38: Discrete Markov Process Defined by the Optimal Policy Vector

$(A_3, D_3) \rightarrow (A_1, D_1)$. This means that the application processing center environment always needs to use design D_1 when processing applications A_1, design D_2 when processing applications A_2, and design D_3 when processing applications A_3. Therefore, whenever there is a change in applications that need to be executed there is a change in design. For this example with these cost values, it is beneficial to change the distributed database design when ever there is a change applications to be processed.

8.7 Summary

In this chapter an approach to generate an optimal redesign policy so as to execute all the applications efficiently in an application processing center environment based on a distributed relational database system has been presented. This integrated approach takes as input the three matrices: probability of change in classes of applications to be processed (P), the matrix giving the cost of redesign (C), and the matrix giving the cost of using a distributed database design for a class of applications (R), and generates as its output an optimal policy vector that dictates the policy to be adopted for each change in class of applications to be processed.

Some guidelines have been developed to define the classes of applications, the calculation of probabilities of change, identifying the change points and execution durations, and generating the set of candidate distributed database designs. These guidelines have been developed so that they can be enforced by analyzing the statistics collected by monitoring the applications being processed and/or using the statistics collected by the distributed database system. The business policies inherent in business data processing are also taken into consideration while defining the classes of applications and deriving the probabilities of change. These guidelines are intuitive in nature and have not been formally defined or verified. This has been left for the future work.

Finally, in this chapter an example with 3 applications and 3 designs is considered and an optimal policy vector is generated. The policy is to use the most optimal design for that class of applications when executing it. In the Appendix B the details of this approach are given.

CHAPTER 9

MARKETING AND RECRUITING SYSTEM: A CASE STUDY

9.1 General Comments

This case study is undertaken to illustrate the distributed database design/redesign methodology proposed in the previous chapters. The Office of Information Technology at Georgia Tech has been developing a student and financial information system based on a relational database system. There is a plan to "connect" the data from a variety of sources into a single distributed database system environment. The proposed design would help in deciding how to fragment and allocate existing relations. The marketing and recruiting system is a module of this information system and consists of a set of applications. The applications defined in this case study are conceptualized from this marketing and recruiting system. This system is used by Georgia Tech personnel to recruit quality students to their undergraduate and graduate programs. These personnel are henceforth known as Recruiters, and the students being recruited are known as Prospects. There are different events held at different places by the institute during the calendar year to inform the prospects about the institute and recruit them.

9.2 Distributed Database Environment

There are five nodes in the distributed database environment. Each node has a computer, and a distributed relational database management system. The nodes are College of Computing (CoC), College of Engineering (CoE), College of Management (CoM), College of Science (CoS), and the Central Administrative Building (CAB). Each of these colleges have Recruiters and maintain the information about the Prospects. All the five nodes are fully connected by a campus wide LAN. All the five nodes have the same computer, with same MIPS rating, Disk I/O time, and the DBMS. This environment with a set of applications accessing the relations defines the application processing center scenario.

9.3 Database Relations and Application

The following set of relations (shown in Tables 10, 11, 12, and 13) are used to store data about Prospects, Recruiters, Events, and relationships among them. These relations are subject to fragmentation and allocation based on the transaction characteristics of the applications accessing the relations. The relation ProsRecr stores data about the relationship between Recruiters and Prospects. The columns of these relations are self explanatory. The sizes of the columns are also given. There is a brief comment about the data items stored in the columns.

Table 10: Relation Recruiters

Column No	Column Name	Size	Comment
c1	Recr–id	Char(9)	Unique Identifier
c2	Recr–type	Char(4)	Type
c3	Recr–name	Char(40)	Name
c4	Recr–add	Char(250)	Address
c5	Recr–major	Char(10)	Major for which the Prospects are recruited
c6	Recr–college	Char(30)	College the recruiter represents
c7	Recr–comments	Char(1024)	Any comments about the Recruiter
c8	Recr–date	Char(10)	Date on which Recruiter started recruiting

Here is the list of perpetual applications which are executed on the above set of relations. These applications are on-line applications that are executed between 8AM and 2PM.

P1. Access Recruiter Information.

P2. Access Prospect Information.

P3. Access Event Calendar.

Here is the list of non–perpetual applications executed on the above set of relations. The applications NP1-NP6 are on-line applications, and the applications NP7-NP12 are batch applications.

Table 11: Relation Prospects

Column No	Column Name	Size	Comment
c9	Pros–id	Char(9)	Unique Identifier
c10	Pros–type	Char(4)	Type 'ug' or 'grad'
c11	Pros–name	Char(40)	Name
c12	Pros–sex	Char(1)	Sex
c13	Pros–ms	Char(1)	Marital Status
c14	Pros–state	Char(2)	State of Residency
c15	Pros–add	Char(250)	Address
c16	Pros–place	Char(60)	Place of residence
c17	Pros–dob	Char(10)	Date of Birth
c18	Pros–resi	Integer	Residency achieved flag
c19	Pros–app–id	Char(7)	Application Identifier
c20	Pros–term–yr	char(10)	Term and Year applied for
c21	Pros–college	Char(30)	College the prospect is interested in
c22	Pros–gpa	Float	Cumulative GPA
c23	Pros–sat–verb	Integer	SAT Verbal score
c24	Pros–sat–quan	Integer	SAT Quantitative score
c25	Pros–sat–anal	Integer	SAT Analytical score
c26	Pros–gre–verb	Integer	GRE Verbal score
c27	Pros–gre–quan	Integer	GRE Quantitative score
c28	Pros–gre–anal	Integer	GRE Analytical score
c29	Pros–gmat	Integer	GMAT score
c30	Pros–toefl	Integer	TOEFL score
c31	Pros–international	Char(1)	Whether out of country

NP1. Match Recruiters to the Prospects (Tuesday and Thursday, 2PM to 4PM).

NP2. List Prospects to whom Information Packets need to be sent (Monday, Wednesday, and Friday 8AM to 10AM).

NP3. Maintain Prospect Information: that is, to update the prospect information, delete or insert prospects. (Every Day 2PM to 5PM).

NP4. Maintain Recruiter Information: that is, to update the recruiter information, delete or insert recruiters (Every Day 2PM to 5PM).

NP5. List Prospects to whom Birthday Greeting Card has to be sent (Everyday 8AM to

Table 12: Relation Events

Column No	Column Name	Size	Comment
c32	Event–id	Char(9)	Unique Identifier
c33	Event–name	Char(40)	Name
c34	Event–type	Integer	Type
c35	Event–place	Char(60)	Place held
c36	Event–state	Char(2)	State of Place
c37	Event–st–date	Char(10)	Start Date
c38	Event–end–date	Char(10)	End Date
c39	Event–max–recr	Integer	Maximum number of recruiters
c40	Event–max–pros	Integer	Maximum number of prospects

Table 13: Relation ProsRecr

Column No	Column Name	Size	Comment
c41	Pros–id	Char(9)	
c42	Recr–id	Char(9)	
c43	Adv–st–date	Char(10)	Start date of advising
c44	Adv–fn–date	Char(10)	Last date of advising
c45	Num–recr	Integer	Number of recruiter

10AM).

NP6. Schedule Events for Prospects and Recruiters (Every Day 10 AM to 12 PM).

NP7. Insert or Update SAT scores (Monday, Wednesday, and Friday 5PM to 8PM).

NP8. Insert or Update GRE scores (Monday, Wednesday, and Friday 5PM to 8PM).

NP9. Insert or Update TOEFL scores (Monday, Wednesday, and Friday 5PM to 8PM).

NP10. Insert or Update GMAT scores (Monday, Wednesday, and Friday 5PM to 8PM).

NP11. Evaluate all the Prospects (Tuesday and Thursday 5PM to 8PM).

NP12. Evaluate all the Recruiters (Tuesday and Thursday 5PM to 8PM).

The SELECT statements defining the tuples and columns accessed by each of the transactions initiated by above applications are given in the Appendix C. There are a total of 70 transactions (T1-T68) which are initiated from the five sites of the distributed database environment.

9.3.1 Defining Classes of Applications

The information about the applications, and the SQL statements initiated by these applications, along with times when these applications are initiated are used to generate the classes of applications. Table 14 lists the applications executed and the transactions[1] initiated by these applications during each hour of the day.

Table 14: Schedule for Applications and Transactions Executed During the Day

Time	Applications Executed	Transactions Initiated
8AM – 9AM	P1, P2, P3, NP2, NP5	T1–T35, T39, T50
9AM – 10AM	P1, P2, P3, NP2, NP5	T1–T35, T39, T50
10AM – 11AM	P1, P2, P3, NP6	T1–T35, T51
11AM – 12AM	P1, P2, P3, NP6	T1–T35, T51
12AM – 1PM	P1, P2, P3	T1–T35
1PM – 2PM	P1, P2, P3	T1–T35
2PM – 3PM	NP1, NP3, NP4	T36–T38, T40–T49
3PM – 4PM	NP1, NP3, NP4	T36–T38, T40–T49
4PM – 5PM	NP3, NP4	T40–T49
5PM – 6PM	NP7–NP12	T51–T68
6PM – 7PM	NP7–NP12	T51–T68
7PM – 8PM	NP7–NP12	T51–T68

In this case study all the applications are grouped into two classes of on-line applications and a class of batch applications. The on-line applications are executed from 8AM to 5PM, and the batch applications from 5PM to 8PM. The two classes of on-line applications are (1) those that are executed from 8AM to 2PM (A1), and (2) those that are executed from 2PM to 5PM (A2). This classification of applications was done because the data and processing requirements of these three classes of applications are different and these applications are executed at different periods of time. The applications classes are as follows.

[1] For the descriptions of transactions, see Appendix C

(A1). P1, P2, P3, NP2, NP5, NP6.

(A2). NP1, NP3, NP4.

(A3). NP7, NP8, NP9, NP10 , NP11, NP12.

At any time of the day one of these classes of applications are executed in the distributed database environment. The schedule of executing the applications is A1 → A2 → A3 → A1.

9.3.2 Transaction and Relation Characteristics

Table 15 gives the cardinalities of the relations in the distributed database environment.

Table 15: Cardinalities of the Relations

Relation	Cardinality
Recruiters	200
Prospects	40000
Events	100
ProsRecr	40000

Table 16 gives the predicates used to access the relations and the approximate cardinality of the tuples satisfying the predicates.

Table 17 gives the frequency of the transactions. The batch applications generate more transactions/sec than on-line applications. Also applications which are active throughout the day initiate fewer transactions/sec than applications which are active for brief intervals of time. In order to use a standard unit of time interval to calculate frequency, for some transactions the values of frequency are less than one. As some of these applications are running concurrently, the total arrival rate of transactions will be the sum of frequencies of all transactions initiated by these applications. This frequency of transactions will be used to come up with the distributed database designs.

Table 16: Selectivity of Predicates in the Relations

Relation Name	Predicate	Cardinality
Recruiters	Recr–type = 'ug'	150
	Recr–type = 'grad'	50
	Recr–college = 'CoC'	30
	Recr–college = 'CoE'	80
	Recr–college = 'CoS'	60
	Recr–college = 'CoM'	30
Prospects	Pros–type = 'ug'	30000
	Pros–type = 'grad'	10000
	Pros–college = 'CoC'	6000
	Pros–college = 'CoE'	16000
	Pros–college = 'CoS'	12000
	Pros–college = 'CoM'	6000
	Pros–state = 'GA'	20000
	Pros–gpa > 3.00	15000
	Pros–gpa > 3.25	8000
	Pros–international = 1	3000
	Pros–toefl > 600	1500
Events	Event–type = 'ug'	70
	Event–type = 'grad'	30

Table 17: Transaction Initiation Frequencies

Frequency/sec	Transactions
0.005	T1–T33
0.05	T34–T35, T41–T48
0.5	T36–T38, T56–T68
1.0	T39, T40, T49–T51
2.0	T52–T55

9.4 Distributed Database Designs

The vertical and horizontal fragmentation algorithms are executed using the transaction and relation characteristics given in the previous section. The distributed database designs for each of the relations are presented below.

9.4.1 Prospects

In order to efficiently execute the class (A1) of applications the following set of vertical fragments are generated by the graphical algorithm presented in [NR89]. By using the graph theoretic algorithm for generating horizontal fragments presented in Chapter 3 following set of horizontal fragments are generated. This was done by (1) generating the predicate usage matrix from the transaction characteristics of the applications A1, (2) generating the predicate affinity matrix from the predicate usage matrix, (3) using the graph theoretic algorithm to generate the initial horizontal fragments, and (4) use the ADJUST function to generate non-overlapping horizontal fragments.
The vertical fragments are:

1. Pros–id, Pros–name, Pros–sex, Pros–ms, Pros–add, Pros–dob.
2. Pros–id, Pros–resi, Pros–app–id, Pros–term–yr.
3. Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal.
4. Pros–id, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal.
5. Pros–id, Pros–gpa, Pros–gmat.
6. Pros–id, Pros–type, Pros–college, Pros–state, Pros–place, Pros–toefl, Pros–international.

The horizontal fragments are:

- **a.** Pros–type = ‘ug’ and Pros-college = ‘CoC’.
- **b.** Pros–type = ‘grad’ and Pros-college = ‘CoC’.
- **c.** Pros–type = ‘ug’ and Pros-college = ‘CoS’.
- **d.** Pros–type = ‘grad’ and Pros-college = ‘CoS’.
- **e.** Pros–type = ‘ug’ and Pros-college = ‘CoE’.
- **f.** Pros–type = ‘grad’ and Pros-college = ‘CoE’.
- **g.** Pros–type = ‘ug’ and Pros-college = ‘CoM’.
- **h.** Pros–type = ‘grad’ and Pros-college = ‘CoM’.

Table 18: Grid Cells of Relation Prospects with Online-Transactions (A1).

	Vertical Fragments					
Horizontal Fragments	**1.**	**2.**	**3.**	**4.**	**5.**	**6.**
a.	T15, T31–T33 T39, T50–T51	T16	T17			
b.	T15, T31–T33 T39, T50–T51	T16		T18		
c.	T19, T31–T33 T39, T50–T51	T20	T21			
d.	T19, T31–T33 T39, T50–T51	T20		T22		
e.	T23, T31–T33 T39, T50–T51	T24	T25			
f.	T23, T31–T33 T39, T50–T51	T24		T26		
g.	T27, T31–T33 T39, T50–T51	T28	T29			
h.	T27, T31–T33 T39, T50–T51	T28			T30	

The set of grid cells generated by these horizontal and vertical fragments, along with the transactions accessing them is presented in Table 18. The analysis of the multiple grid cells being accessed by the transactions generates $\mathcal{T}^{(2)} = \{f_1^2 = (1_{(a,b)}), f_2^2 = (1_{(c,d)}), f_3^2 = (1_{(e,f)}), f_4^2 = (1_{(g,h)}) f_5^2 = (2_{(a,b)}), f_6^2 = (2_{(c,d)}), f_7^2 = (2_{(e,f)}), f_8^2 = (2_{(g,h)})\}$, and $\mathcal{T}^{(8)} = \{f_1^8 = (1_{(a,b,c,d,e,f,g,h)}\}$ (refer to Section. 4.2.2).

The procedures described for grid optimization in Section 4.3 are used to merge the grid cells to form mixed fragments. The Grid_ Optimization() algorithm is used to generate the following set of mixed fragments. This was done by (1) calculating the sizes of grid cells from the lengths of columns of the Recruiters relation, and the selectivity of the predicates accessing the tuples, (2) calculating the number of data accesses required to access the grid cells and merged fragments, and (3) comparing these number of data accesses to decide on whether or not to merge.

Note that the mixed fragments defined by grid cells represented by sets f_1^2, f_2^2, f_3^2, and f_4^2 are contained in the mixed fragment defined by f_1^8. Hence there is a need to evaluate

whether it is beneficial to have a single mixed fragment defined by grid cells in f_1^8 or four mixed fragments defined by grid cells in sets f_1^2, f_2^2, f_3^2, and f_4^2. This evaluation is done by using the cost model model presented in section 4.3.1 and the evaluation algorithm Eval_contained_in(), presented in Section 4.3.4. For simplicity it is assumed that all transactions access the grid cells by using the segment scan with a *Pre Fetch Blocking Factor* as 1, and a page size of 1K bytes. The cardinalities of horizontal fragments a, b, c, d, e, f, g, h are calculated from the Table 16. The lengths of the vertical fragments $1, 2, 3, 4, 5, 6$ are calculated from the lengths of the columns of the relation Prospects given in Table 11. Based on the above data about sizes of grid cells and mixed fragments, the number of disk accesses to process all the transactions accessing the mixed fragments f_1^2, f_2^2, f_3^2, and f_4^2 is 41676, and to process all the transactions accessing the mixed fragment f_1^8 is 41883, therefore it is beneficial to break up mixed fragment f_1^8 to four mixed fragments represented by the sets of grid cells f_1^2, f_2^2, f_3^2, and f_4^2. But as the transactions T50 and T51 which access these mixed fragments with a high frequency of 1.0/sec are initiated from site CAB, all these mixed fragments defined by $f_1^2, f_2^2, f_3^2, f_4^2$ are allocated to site CAB.

Next we evaluated the benefit of having mixed fragments defined by grid cells in sets f_5^2, f_6^2, f_7^2, and f_8^2, against a single mixed fragment formed by represented by $2_{(a,b,c,d,e,f,g,h)}$. It was found that both of them result in same number of data accesses (162) to process all the transactions accessing the grid cells $\{2_a, 2_b, 2_c, 2_d, 2_e, 2_f, 2_g, 2_h\}$. In order to increase the locality of processing for the transactions accessing the grid cells, set of mixed fragments were formed by merging the grid cells in each of the sets f_5^2, f_6^2, f_7^2, and f_8^2. The resulting mixed fragments are allocated to sites CoC, CoS, CoE and CoM respectively.

The vertical fragments $3, 4, 5$ are accessed at most by one transaction each. Therefore, they were merged to form a mixed fragment represented by grid cells $\{(3_{(a,b,c,d,e,f,g,h)}$, $4_{(a,b,c,d,e,f,g,h)}$, $5_{(a,b,c,d,e,f,g,h)})\}$. Otherwise there will be too many fragments for the relation, and it will increase the cost of maintaining the distributed database. The vertical fragment 6 which is not accessed by any of the transactions T1-T68 is merged to form a mixed fragment $\{(6_{(a,b,c,d,e,f,g,h)})\}$. These two mixed fragments are allocated to site CAB. The distributed database design for relation Prospects to execute the class of applications (A1) is shown in Table 19.

In order to efficiently process the transactions initiated by class (A2) of on-line applications, following sets of vertical and horizontal fragments were generated.

Table 19: Distributed Database Design for relation Prospects (A1)

Application	Relation	Fragment Id	Fragment Name	Site
On-Line Class A1	Prospects	p_1	$(1_{(a,b)})$	CAB
		p_2	$(1_{(c,d)})$	CAB
		p_3	$(1_{(e,f)})$	CAB
		p_4	$(1_{(g,h)})$	CAB
		p_5	$(2_{(a,b)})$	CoC
		p_6	$(2_{(c,d)})$	CoS
		p_7	$(2_{(e,f)})$	CoE
		p_8	$(2_{(g,h)})$	CoM
		p_9	$(3_{(a,b,c,d,e,f,g,h)}, 4_{(a,b,c,d,e,f,g,h)}$ $5_{(a,b,c,d,e,f,g,h)})$	CAB
		p_{10}	$(6_{(a,b,c,d,e,f,g,h)})$	CAB

The set of vertical fragments are:

1′. Pros–id, Pros–name, Pros–sex, Pros–ms, Pros–add, Pros–state, Pros–resi, Pros–app–id, Pros–term–yr.

2′. Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal.

3′. Pros–id, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal.

4′. Pros–id, Pros–gpa, Pros–gmat.

5′. Pros–id, Pros–type, Pros–college, Pros–place, Pros–dob, Pros–toefl, Pros–international.

The set of horizontal fragments are the same as those for the class of on-line application A1.
The set of grid cells generated by these horizontal and vertical fragments, along with the transactions accessing them is presented in Table 20.

By executing the Grid_Optimization() algorithm it was found beneficial to merge grid cells represented by $\{1'_{(a,b,c,d,e,f,g,h)}\}$ as a mixed fragment and allocate it to the site CAB. It was also found beneficial to merge grid cells represented by $\{(2'_{(a,b,c,d,e,f,g,h)}, 3'_{(a,b,c,d,e,f,g,h)}, 4'_{(a,b,c,d,e,f,g,h)})\}$, and $\{(5'_{(a,b,c,d,e,f,g,h)})\}$ as mixed fragments and allocate them to the site CAB. The distributed database design for relation Prospects to execute the class of on-line applications (A2) is shown in Table 21.

Table 20: Grid Cells of Relation Prospects with Online-Transactions (A2).

Horizontal Fragments	Vertical Fragments 1′.	2′.	3′.	4′.	5′.
a.	T40	T41			
b.	T40		T42		
c.	T40	T43			
d.	T40		T44		
e.	T40	T45			
f.	T40		T46		
g.	T40	T47			
h.	T40			T48	

Table 21: Distributed Database Design for relation Prospects (A2)

Application	Relation	Fragment Id	Fragment Name	Site
On-Line Class A2	Prospects	p'_1	$1'_{(a,b,c,d,e,f,g)}$	CAB
		p'_2	$(2'_{(a,b,c,d,e,f,g,h)}, (3'_{(a,b,c,d,e,f,g,h)}$ $4'_{(a,b,c,d,e,f,g,h)})$	CAB
		p'_3	$(5'_{(a,b,c,d,e,f,g,h)})$	CAB

For the class (A3) of batch applications the set of vertical and horizontal fragments that efficiently process all the transactions are given below.
The vertical fragments are:

1″. Pros–id, Pros–name, Pros–add.

2″. Pros–id, Pros–sex, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal.

3″. Pros–id, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal.

4″. Pros–id, Pros–gmat.

5″. Pros–id, Pros–toefl, Pros–international.

6″. Pros–id, Pros–type, Pros–ms, Pros–state, Pros–place, Pros–dob, Pros–app–id, Pros–term–yr, Pros–college.

The horizontal fragments are the same as that for the classes { (A1), (A2)} of on-line applications. Table 22 shows the grid cells and the transactions accessing these grid cells.

Table 22: Grid Cells of Relation Prospects with Batch Transactions

	Vertical Fragments					
Horizontal Fragments	1″.	2″.	3″.	4″.	5″.	6″.
a.	T52, T66	T56, T58			T54	
b.	T53, T67		T56a, T59		T54	
c.	T52, T66	T56,T60			T54	
d.	T53, T67		T56a,T61		T54	
e.	T52, T66	T56, T62			T54	
f.	T53, T67		T56a, T63		T54	
g.	T52, T66	T56, T64			T54	
h.	T53, T68			T56a, T65, T68	T54	

By executing the Grid_Optimization() algorithm it was found beneficial to form mixed fragments ($1''_{(a,c,e,g)}$), ($1''_{(b,d,f,h)}$), ($5''_{(a,b,c,d,e,f,g,h)}$). The rest of the grid cells were merged to form mixed fragments ($2''_{(a,b,c,d,e,f,g,h)}, 3''_{(a,b,c,d,e,f,g,h)}, 4''_{(a,b,c,d,e,f,g,h)}$), and ($6''_{(a,b,c,d,e,f,g,h)}$). Since all the processing on these fragments is initiated from the site CAB, all these fragments are allocated at the site CAB. The distributed database design for relation Prospects to execute the class of applications (A3) is shown in Table 23.

Table 23: Distributed Database Design for relation Prospects (A3)

Application	Relation	Fragment Id	Fragment Name	Site
Batch	Prospects	p''_1	($1''_{(a,c,e,g)}$)	CAB
		p''_2	($1''_{(b,d,f,h)}$)	CAB
		p''_3	($5''_{(a,b,c,d,e,f,g,h)}$)	CAB
		p''_4	($2''_{(a,b,c,d,e,f,g,h)}, 3''_{(a,b,c,d,e,f,g,h)}$ $4''_{(a,b,c,d,e,f,g,h)}$)	CAB
		p''_5	($6''_{(a,b,c,d,e,f,g,h)}$)	CAB

9.4.2 Recruiters

Similarly, for efficiently executing the class (A1) of on-line applications, the vertical and horizontal fragmentation algorithms generate the following set of vertical and horizontal fragments.

1. Recr–id, Recr–name, Recr–add, Recr–type.

2. Recr–id, Recr–major, Recr–college, Recr–comment, Recr–date.

a. Recr–type = 'ug' and Recr-college = 'CoC'.

b. Recr–type = 'grad' and Recr-college = 'CoC'.

c. Recr–type = 'ug' and Recr-college = 'CoS'.

d. Recr–type = 'grad' and Recr-college = 'CoS'.

e. Recr–type = 'ug' and Recr-college = 'CoE'.

f. Recr–type = 'grad' and Recr-college = 'CoE'.

g. Recr–type = 'ug' and Recr-college = 'CoM'.

h. Recr–type = 'grad' and Recr-college = 'CoM'.

The set of grid cells generated by these horizontal and vertical fragments along with the transactions accessing them is presented in the Table 24.

Note that none of the transactions T1–T68 access the vertical fragment 2. The Grid_Optimization() algorithm is used to generate the following set of mixed fragments. These mixed fragments are specified by means of the representation scheme as, $f_1 = 1_{(a,b)}$, and $f_2 = 1_{(c,d)}$, and $f_3 = 1_{(e,f)}$, and $f_4 = 1_{(g,h)}$, and $f_5 = 2_{(a,b,c,d,e,f,g,h)}$. The mixed fragments f_1, f_2, f_3, and f_4 are respectively located at sites CoC, CoS, CoE, and CoM. And the mixed fragment f_5 is located at site CAB. The distributed database design for relation Recruiters when executing class of on-line applications (A1) is shown in Table 25.

For efficiently executing the class (A2) of on-line applications the vertical and horizontal fragmentation algorithms generate the following set of vertical and horizontal fragments. The vertical fragments are:

$1'$. Recr–id,Recr–College.

Table 24: Grid Cells of Relation Recruiters for Class of Online-Transactions (A1).

	Vertical Fragments	
Horizontal Fragments	**1.**	**2.**
a.	T1, T2, T13	
b.	T1, T3, T14	
c.	T4, T5, T13	
d.	T4, T6, T14	
e.	T7, T8, T13	
f.	T7, T9, T14	
g.	T10, T11, T13	
h.	T10, T12, T14	

Table 25: Distributed Database Design for relation Recruiters (A1)

Application	Relation	Fragment Id	Fragment Name	Site
On-Line Class A1	Recruiters	r_1	$(1_{(a,b)})$	CoC
		r_2	$(1_{(c,d)})$	CoS
		r_3	$(1_{(e,f)})$	CoE
		r_4	$(1_{(g,h)})$	CoM
		r_5	$(2_{(a,b,c,d,e,f,g,h)})$	CAB

$2'$. Recr–id, Recr–type, Recr–name, Recr–add, Recr–major, Recr–comments, Recr–date.

The horizontal fragments are:

a'. Recr–college = ‘CoC’.

b'. Recr–college = ‘CoS’.

c'. Recr–college = ‘CoE’.

d'. Recr–college = ‘CoM’.

The set of grid cells generated by this design along with the transactions accessing them is presented in Table 26.

Table 26: Grid Cells of Relation Recruiters for Class Online-Transactions (A1).

Horizontal Fragments	Vertical Fragments $1'$.	$2'$.
a'.	T36 ,T49	T49
b'.	T37a ,T49	T49
c'.	T37 ,T49	T49
d'.	T38 ,T49	T49

Table 27: Distributed Database Design for relation Recruiters (A2)

Application	Relation	Fragment Id	Fragment Name	Site
On-Line Class A2	Recruiters	r'_1	$(1'_{(a',b',c',d')}, 2'_{(a',b',c',d')})$	CAB

The frequency with which transaction T49 is initiated is much higher than transactions T36-T38. Therefore, the Grid_Optimization() algorithm generates a single mixed fragment by merging all the grid cells. This mixed fragment represented by $r'_1 = (1'_{(a',b',c',d')}, 2'_{(a',b',c',d')})$ is allocated at the site CAB. The distributed database design for relation Recruiters to execute class of applications (A2) is shown in Table 27.

For efficiently executing the class (A3) of batch applications the vertical and horizontal fragmentation algorithms generate the following set of vertical and horizontal fragments.

$1''$. Recr–id, Recr–name, Recr–college, Recr–type.

$2''$. Recr–id, Recr–major, Recr–add, Recr–comment, Recr–date.

The horizontal fragments are the same as that for the class (A1) of on-line applications. The grid cells and the transactions accessing them for the class (A3) of batch applications are given Table 28.

Note that none of the transactions T1-T68 access the vertical fragment $2''$. The Grid_Optimization() algorithm generated the following set of mixed fragments represented by $f''_1 = 1''_{(a,c,e,g)}$, $f''_2 = 1''_{(b,d,f,h)}$, and $f''_3 = 2''_{(a,b,c,d,e,f,g,h)}$. Again in this case also all these fragments are allocated at site CAB, as the relation Recruiter has small relation size, and the transactions accessing the relation are all initiated from site CAB. The distributed

Table 28: Grid Cells of Relation Recruiters with Batch-Transactions

Horizontal Fragments	Vertical Fragments 1″.	2″.
a.	T66	
b.	T67	
c.	T66	
d.	T67	
e.	T66	
f.	T67	
g.	T66	
h.	T67	

Table 29: Distributed Database Design for relation Recruiters (A3)

Application	Relation	Fragment Id	Fragment Name	Site
Batch	Recruiters	r''_1	$1''_{(a,c,e,g)}$	CAB
		r''_2	$1''_{(b,d,f,h)}$	CAB
		r''_3	$2''_{(a,b,c,d,e,f,g,h)}$	CAB

database design for relation Recruiters to execute the class of applications (A3) is shown in Table 29.

9.4.3 Other Relations

The relation Events has a small cardinality and is allocated at site CAB. The relation ProsRecr is accessed by very few transactions from the site CAB. Therefore, it was not beneficial to fragment relation ProsRecr. The relation ProsRecr was allocated to the site CAB as all the transactions that accessed it were initiated from that site.

9.4.4 Fragmentation and Allocation Schemes for On-line and Batch Applications

Figures 39, 40, and 41 show the location of the fragments, the transactions initiated from each of the sites in the distributed database environment, and the optimal distributed

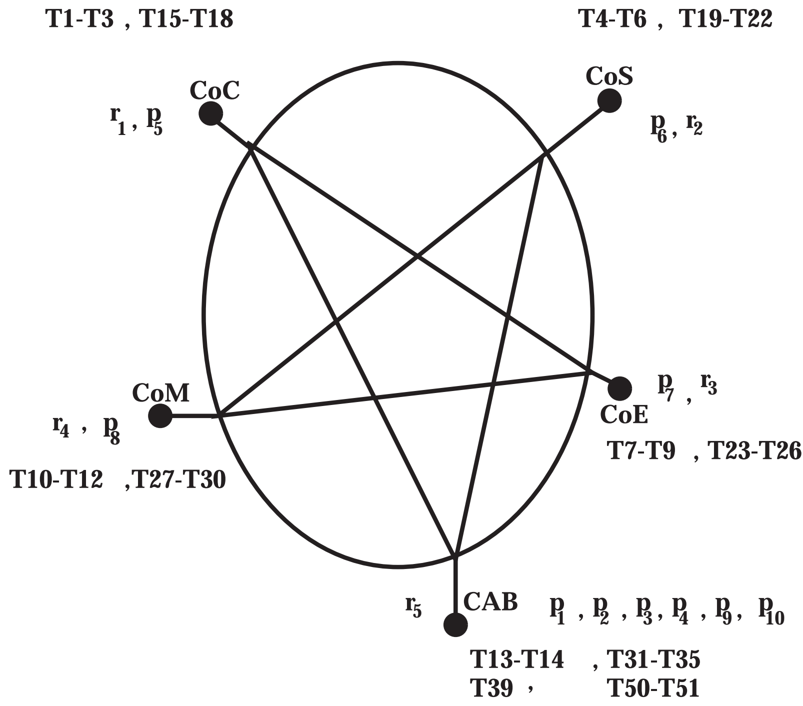


Figure 39: Candidate Distributed Database Design (D1) for On-Line Applications (A1).

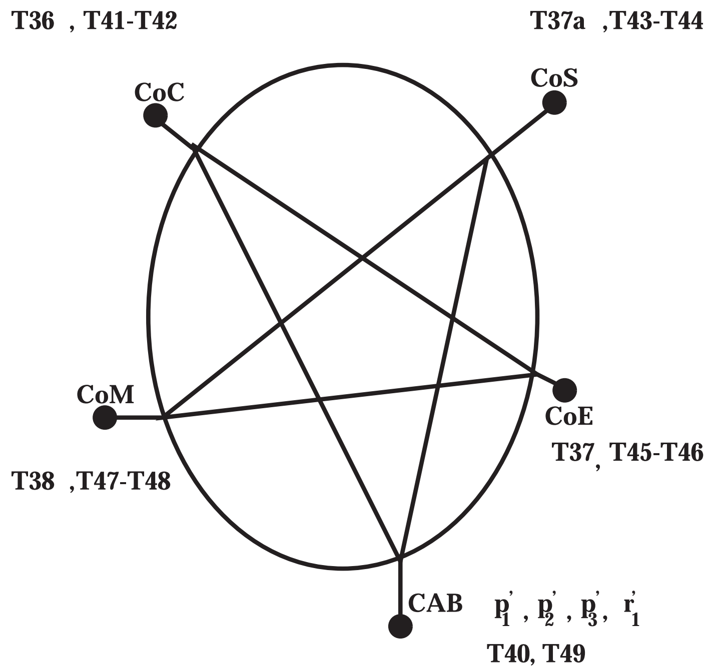


Figure 40: Candidate Distributed Database Design (D2) for On-Line Applications (A2).

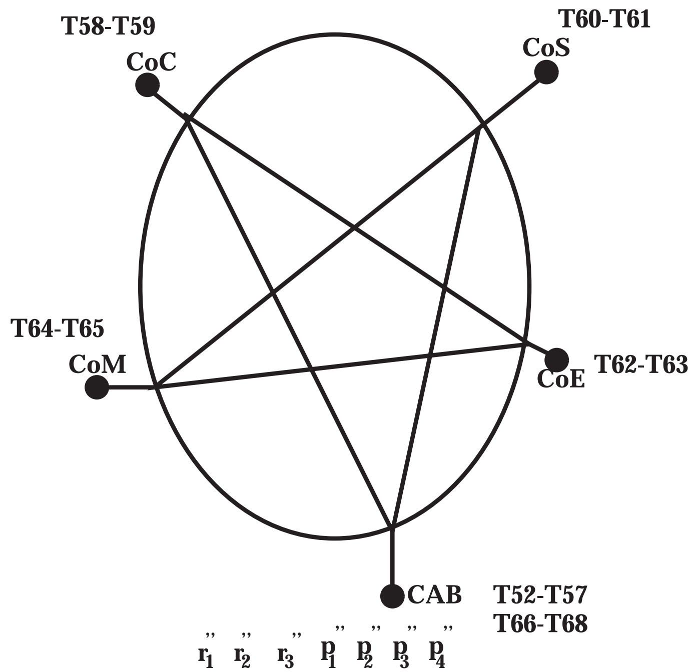


Figure 41: Candidate Distributed Database Design (D3) for Batch Applications (A3).

database design (D1) for the class (A1) of online applications, optimal distributed database design (D2) for class (A2) of on-line applications, and optimal distributed database design (D3) for class (A3) of batch applications respectively. From these optimal distributed database designs following points are noted:

- Only for the optimal design (D1) for the class (A1) of on-line applications fragments are allocated at more than one site.
- There is a change only in fragmentation scheme and not in the allocation scheme between optimal distributed database designs (D2) and (D3). This is because different sets of transactions access different columns of the relations. That is, the vertical fragmentation scheme changes, but the allocation scheme does not change.
- All the fragments of the optimal distributed database designs (D2) and (D3) are allocated at site CAB because the transactions initiated from the site CAB access all these fragments with a high frequency.

9.5 Application Processing Center Scenario

There are three classes of applications that are executed in the distributed database environment. They are two classes of on-line applications { (A1), (A2)} and the class of batch

applications (A3). In the previous section three optimal distributed database designs, one each for on-line applications { (A1), (A2)}, and another for batch applications (A3) were generated by using the methodology presented in Chapters 3, 4.

From the Figures 39, 40, and 41, and Tables 24,26,28, 18, 20, and 22 the percentage of local and distributed transactions initiated from each site are calculated. Moreover, for each distributed transaction the number of sites from which it accesses the data is calculated. Since these three classes of applications that follow a strict schedule for their execution $A1 \rightarrow A2 \rightarrow A3 \rightarrow A1$, the probability of change in class of applications to be executed at the next change point is given by the stochastic matrix P.

$$P = \begin{bmatrix} 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 1.0 & 0.0 & 0.0 \end{bmatrix}$$

9.6 Cost Values Calculation

In chapter 8 the cost values needed for the Markovian decision analysis were presented. There were two cost values defined, (1) the cost of executing a class of applications on a candidate distributed database design, and (2) the cost of redesign. In this case study the cost of executing a class of applications on a distributed database design is the average transaction response time estimated through simulation. There are three classes of applications, and three candidate distributed database designs. The classes of applications are on-line and batch. The first two candidate distributed database designs (D1) and (D2) are optimal for the classes of on-line applications (A1) and (A2) respectively. The third candidate distributed database design (D3) is optimal for the class of batch applications (A3). There are nine cost values representing the estimated average transaction response time by simulating the execution of batch applications (A3) on each of the three candidate designs, and by simulating the execution of the two classes of on-line applications {(A1), (A3)} on each of the three candidate designs. These nine cost values were estimated as follows:

1. All the transactions executed by each class of applications are mapped to the sites from which they are initiated.

2. For each transaction initiated from each site the following information is tabulated:

- Whether it is distributed or local transaction: if distributed, the number of sites involved in processing the transaction.
- Number of fragments accessed by the transaction.
- For transactions accessing more than two fragments, whether a distributed join, or a distributed union, or a local join, or a local union is performed.

3. Average number of data accesses for a local and distributed transaction are estimated based on above information. In an implemented system, this information can be gathered from the database system utilities, and transaction trace analysis.

4. At each site the arrival rate of local and distributed transactions are calculated by using the frequency information from Table 17. From this average arrival rate of local and distributed transactions at each site is calculated.

5. Based on the estimates for data accesses for each transaction, average number of data accesses for local and distributed transactions are calculated.

6. The average data access time is taken to be 50 milliseconds and the average communication delay is taken to be 0.1 seconds. The total number of locks in the distributed database environment are estimated to be 30,000. This is based on the size of the fragments, and a page size of 1K bytes.

7. The transaction path lengths and the number of initial disk I/O's are same as those used in Chapter 7. The average number of locks are same as the number of data accesses.

8. Finally, nine simulations were run to estimate the average transaction response time for each combination of a class of applications and a candidate distributed database design.

Then the matrix R giving the benefit (negative of cost) values of executing each class of applications on each of the three candidate designs is:

$$R = \begin{bmatrix} -0.703 & -0.924 & -1.232 \\ -1.464 & -1.234 & -1.283 \\ -1.685 & -1.409 & -0.949 \end{bmatrix}$$

In order to calculate the cost of redesign the operator method described in Chapter 5 was used. The data transfer rate was assumed to be 50Kb/sec. This is because the distributed data base environment is based on a local area network with full connectivity. The cost model developed in Section 5.6 was used to calculate the cost of redesign in seconds. The redesign time taken in seconds was:

$$RedesignTimeTaken = \begin{bmatrix} 0 & 3325 & 3975 \\ 3897 & 0 & 3746 \\ 2818 & 2561 & 0 \end{bmatrix}$$

But this time has to be amortized over the number of transactions that are executed in the next time interval. The class of applications (A1) initiate 68364 transactions in six hours, class of applications (A2) initiate 47520 transactions in three hours, and class of batch applications initiate 162000 transactions in three hours.

Therefore, the additional overhead of redesign cost per transaction is given by matrix:

$$C = \begin{bmatrix} 0 & -0.07 & -0.024 \\ -0.06 & 0 & -0.023 \\ -0.04 & -0.053 & 0 \end{bmatrix}$$

Then by using the probability matrix P, the cost matrices R and C, the optimal policy vector $Policy^2$ generated by the Markovian decision analysis algorithms is:

$$Policy = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 3 \\ 3 \\ 3 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Figure 42 shows the long run discrete Markov process (the states being (A1,D1), (A2,D1), and (A3,D3)) defined by the above policy. This policy implies that in the long run it is

[2]Please refer to Table 9 for the meaning of each of the policy numbers in the policy vector.

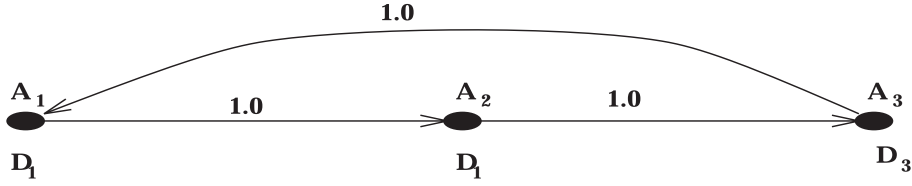


Figure 42: Long Run Discrete Markov Process For Application Processing Scenario I.

always optimal to use distributed database design (D1) for executing the class (A1) of on-line applications and the class (A2) of on-line applications. But for executing the class (A3) of batch applications it is preferred to use the candidate distributed database design (D3). The design (D1) is preferred for processing classes of on-line applications (A1) and (A2) in the long run because it gives the least average transaction response time. For the same reason design (D3) is preferred for the class of batch applications (A3). Thus there is a change in design whenever there is a change from processing the class of on-line applications to processing the class of batch applications and vice versa.

A scenario with randomness in the class of applications initiated is considered next. After executing the class (A1) of applications for six hours, the class (A3) of batch applications are executed for one hour. Similarly, after executing the class (A2) of applications (A2) for three hours, the class (A3) of batch applications are executed for one hour. But after executing the class of batch applications, one of the classes of on-line applications {(A1), (A2)} is executed with equal probability. This is a practical scenario because during the lunch time the class of batch applications are executed for one hour and after the day's work the class of batch applications are again run for one hour. In the mean time any class of on-line applications are executed with equal probability. This scenario incorporates the randomness in the class of applications to be initiated. The probability of change matrix will be given by the following stochastic matrix:

$$P = \begin{bmatrix} 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \\ 0.5 & 0.5 & 0.0 \end{bmatrix}$$

The cost of executing the classes of application on the candidate designs does not change, but the overhead due to the cost of redesign changes. The cost of redesign overhead per transaction for this application scenario is:

$$C = \begin{bmatrix} 0.0 & -0.07 & -0.073 \\ -0.06 & 0.0 & -0.069 \\ -0.04 & -0.053 & 0.0 \end{bmatrix}$$

For this case, the optimal policy vector generated by the Markovian decision analysis was:

$$Policy = \begin{bmatrix} 3 \\ 3 \\ 3 \\ 3 \\ 3 \\ 3 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

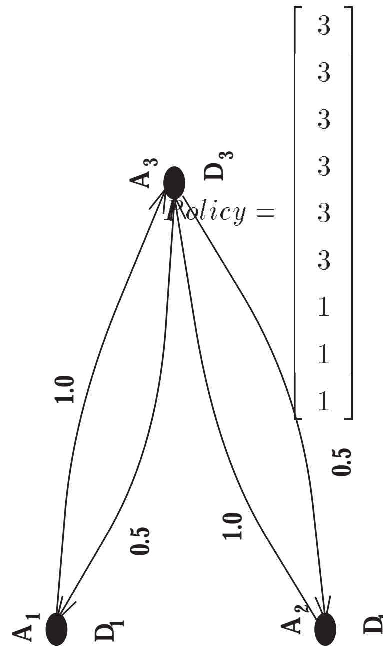


Figure 43: Long Run Discrete Markov Process For Application Processing Scenario II.

The analysis of the above policy vector shows that in the long run (see Figure 43), distributed database design (D2) will be the least costly to use for the classes of on-line applications (A1) and (A2). And the candidate distributed database design (D3) will be the least costly to run the class of batch applications (A3). The optimal policy shows that it is beneficial to do redesign for this case also.

9.7 Summary

The objective of this case study was to illustrate the applicability of the techniques proposed in this thesis to a real life application processing scenario. In this case study, the mixed fragmentation algorithms were used to generate the candidate designs, the simulation model was used to estimate the average transaction response time, and the algorithms developed to materialize the redesigned distributed databases were used to calculate the cost of redesign. The applications, transactions and their characteristics were derived from a similar system currently in use at Georgia Tech. In this case study two application processing scenarios were considered. In the first scenario, the classes of applications were executed in a strict schedule, and in the second scenario, two of the classes of applications were executed in a random manner. For both these scenarios it was shown that the design (D1) was the least costly for processing both the classes {(A1), (A2)} of on-line applications. On the contrary, the optimal design (D3) was the least costly for processing the class (A3) of batch applications. Hence it is beneficial to switch between these designs to process the on-line and the batch applications in the long run. The case study is governed by the characteristics of the applications. As there were no tools or software implemented for generating the candidate distributed database designs, calculating the cost of redesign, and calculating the parameter values for each of the simulation runs, a more extensive case study with lot more applications and distributed database designs could not be undertaken. The applicability of techniques proposed in this thesis to a real life application processing scenario has been illustrated.

CHAPTER 10

CONCLUSIONS

Distributed database environment is a dynamic system; there many physical and logical changes that cause a deterioration in the performance of the applications. The distributed database design improves the performance of the application by reducing the irrelevant data accessed, and data transferred. Therefore, the problem is to redesign the distributed database so as to efficiently process the applications in a dynamic distributed database environment. Redesign can be corrective, preventive, or adaptive. The corrective redesign is the easiest, and adaptive redesign is the toughest. This thesis addresses the problem of preventive redesign where there is a random behavior in the classes of applications being executed in the distributed database environment over a time period. This random behavior is modeled as a discrete Markov process. A set of candidate distributed database designs are generated for a set of classes of applications, and the problem is to select the optimal distributed database design for each execution of an application in the long run. There are two cost values that govern this selection. The first one is the cost of using a particular candidate design by an application. The second one is the cost of materializing the populated redesigned distributed database. This thesis provides a solution for the redesign methodology in this scenario. The major contributions of this thesis are:

- A mixed fragmentation methodology has been presented for the distribute database design problem. This methodology is used to generate all the candidate designs involving a fragmentation of the database and the allocation of these fragments for a given set of classes of applications.

- A representation scheme for grid cells, and the notion of regular fragments has been developed. This representation scheme is used for merging the grid cells into mixed fragments, and materializing distributed databases.

- Two elegant solutions have been developed for the problem of materializing distributed databases. They are the query generator approach and the operator approach. A

cost model to compare these two solutions has been developed. A multiple query optimization algorithm has been developed to increase the efficiency of the query generator approach.

- A simulation model has been presented to estimate the average transaction response time in the distributed relational database environment. The effect of changes in data access time, arrival rate, and the number of data accesses on the average transaction response time has been presented. The tradeoff between the time spent on data access and communication delay has been studied. Application processing cost is to be approximated as the total average cost of processing the transactions it comprises.

- A distributed database redesign methodology based on Markovian decision analysis has been developed. The Markovian decision analysis uses the cost of materialization, the average transaction response time, and the stochastic matrix defining the random behavior in executing the applications to generate an optimal policy. This optimal policy specifies the candidate distributed database design that needs to be used by any application to be executed. This optimal policy is guaranteed to process all the applications efficiently in the long run.

- Finally, a real life case study has been developed by conceptualizing a set of distributed databases and applications in use at Georgia Tech. Currently, there is no single distributed DBMS that interconnects all databases. But we assume for this case study that such a system is in existence. This case study consisted of two classes of on-line applications, and a class of batch applications. The optimal distributed database designs using the mixed fragmentation methodology were generated for each of the classes of applications. Two application processing scenarios were modeled as discrete Markov processes, and the Markovian decision analysis was used to generate the optimal policies. These optimal policies specified that a switching between two distributed database designs is required to efficiently execute the given classes of applications in the long run for both the application processing scenarios.

This is the first piece of work in the total redesign of distributed databases to the best of our knowledge. The solutions presented in this dissertation are subject to further analysis and optimization. Following are the set of problems yet to be addressed in this area of research.

Evaluation of Algorithms to Materialize Distributed Relational Databases.

1. The operator method uses the operations which are specified at higher level like "split a fragment", "merge two fragments", etc. But detailed algorithms specifying how to implement these operations at the storage manager level have not been developed. A particular distributed database management system like client-server versions of Oracle or Sybase or Ingres needs to be considered to design and evaluate the algorithms to perform the basic split, merge and move operations.

2. The design and development of these algorithms should also take into consideration the issues of incremental materialization of redesigned distributed databases while the transactions are accessing the database. This would require scheduling the materialization operations on the fragments of the relations such that a minimal number of transactions that update the database are blocked. In a well defined application processing center scenario it would be feasible to know in advance which transactions are going to be executed. This would enable us to schedule the materialization of fragments that are not being updated by these transactions.

3. Caching of the temporary tables created during materialization process will increase the efficiency of the materialization algorithms. Hence a topic of future research is to study the effect of caching and buffer management on the efficiency of the materialization algorithms.

4. Finally, these algorithms need to be tested on large distributed databases and experimented with various parameters such as size of relations, number of replications, number of sites, number of fragments per relation, etc. This experimentation will give us an insight into the materialization and maintenance problems of large distributed relational databases.

Distributed Database Design/Redesign Tool

In this thesis we have shown the viability of the redesign methodology by means of a case study. The next step will be to design and implement a practical distributed database design and redesign tool. The architecture of the tool is presented in Figure 44. The various modules of the tool are as follows:

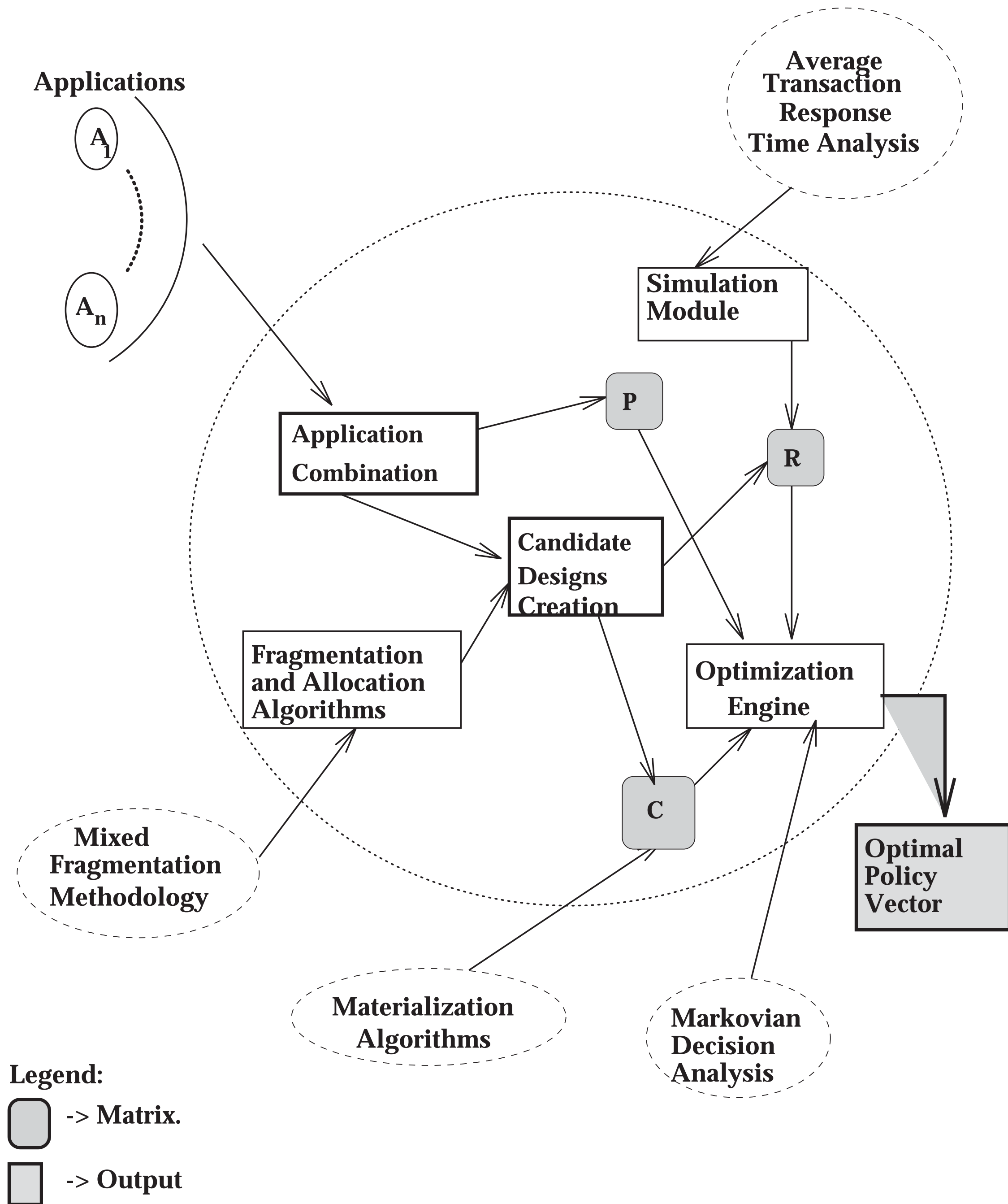


Figure 44: A Distributed Database Design/Redesign Tool Architecture.

- A set of algorithms in the module called Application Combination to form a set of classes of applications from a given set of applications, and generate the stochastic matrix P.

- A set of algorithms implemented to generate the fragmentation and allocation schemes based on the mixed fragmentation methodology. This module is used to generate a set of candidate designs.

- A simulation module to estimate the average transaction response time for executing a class of applications on each of the candidate distributed database designs. This module generates the matrix R.

- A set of algorithms to calculate the cost for materializing distributed databases. These set of algorithms generate the matrix C.

- An optimization engine that takes as input the matrices P, R, and C, and outputs the optimal policy vector.

Extensions to the Redesign Methodology

- The random behavior of execution of classes of applications is not always a discrete Markov process; therefore, this methodology has to be extended to the case when this behavior follows a continuous Markov process or a semi-Markov process.

- The simulation model used to estimate the average transaction response time needs to be extended to incorporate optimistic, and semi-optimistic concurrency control protocols. In fact, the distributed database design tool should be integrated with the distributed database management system so that the average transaction response is directly measured.

- Allocation of mixed fragments depends on the transactions and the sites of their origin, the transaction processing semantics, and the network characteristics. The transaction processing characteristics are dependent on a particular relational database system. Because of these interdependent factors, the allocation of mixed fragments is a very tough problem and needs to be addressed in future.

- The adaptive redesign problem is a very tough problem. In order to attack this problem an adaptive distributed database design methodology based on an active

database needs to be considered. The active database system has the capability of monitoring the statistics collected by the distributed database system so as to initiate actions to change the parameter values for generating the distributed database design. The problem is to identify the set of parameters that can make the distributed database design methodology adaptive to dynamic changes.

- In this thesis we have just considered developing distributed database design/ redesign methodology when there are no inherent constraints placed on the performance of the applications. A tougher problem would be to design a distributed database that guarantees a certain minimum transaction response time. Such problems will arise in distributed multimedia database application processing scenarios, and real time distributed database transaction processing.

Appendix A

BREAK, FORM AND CLUSTER OPERATIONS

Note that the fragments in the fragmentation schemes are all regular. A regular fragment is represented by $\{\alpha_{1A}, \alpha_{2A}, \ldots, \alpha_{nA}\}$. Where $\alpha_1, \alpha_2, \ldots, \alpha_n$ are the basic vertical fragments, and $A = \{a_1, a_2, \ldots, a_n\}$ denote the set of basic horizontal fragments. Then the operation break is defined by the following algorithm.

break(f)

```
begin
repeat for each α_i ∈ r(f)
    split_v(f, α_iA, f');
    v = α_iA;
    repeat for each a ∈ A
        split_h(v, α_ia, f'');
        v = f'';
    end repeat
    f = f';
end repeat
end begin
```

The *form* procedure is analogous to the break procedure. It is given below.

form(f)

```
begin
repeat for each α_i ∈ r(f)
    v = ∅
    repeat for each a ∈ A
```

$merge_h(v, \alpha_{ia}, f');$

$v = f';$

end repeat

$merge_v(v, v', f'')$

$v' = f''$

end repeat

$f = v'$

end begin

The function $cluster(f)$ is defined as follows

cluster(f)

begin

$Cl = \emptyset$

repeat for each $\alpha_i \in r(f)$

repeat for each $a \in A$

$Cl = Cl \bigcup \{\alpha_{ia}\}$

end repeat

end repeat

$Clust(f) = Cl$

end begin

Appendix B

OPTIMAL POLICY VECTOR GENERATION DETAILS

Let the initial policy vector be:

$$Policy = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

That is, irrespective of which class of applications are being executed this policy assigns the distributed database design D_1 to be used. As the above policy defines the discrete markov process, the state transition probability matrix S is given (refer [How60]) as

$$\mathcal{S} = \begin{bmatrix} 0.8 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.2 & 0.0 & 0.0 \\ 0.8 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.2 & 0.0 & 0.0 \\ 0.8 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.2 & 0.0 & 0.0 \\ 0.2 & 0.0 & 0.0 & 0.8 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.2 & 0.0 & 0.0 & 0.8 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.2 & 0.0 & 0.0 & 0.8 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.2 & 0.0 & 0.0 & 0.8 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.2 & 0.0 & 0.0 & 0.8 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.2 & 0.0 & 0.0 & 0.8 & 0.0 & 0.0 \end{bmatrix}$$

The cost processing the applications using this policy is dictated by the reward matrix R, which gives the benefit (-ve cost) for each state transition as:

$$
\mathcal{R} = \begin{bmatrix}
-10.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -10.0 & 0.0 & 0.0 \\
-25.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -25.0 & 0.0 & 0.0 \\
-25.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & -25.0 & 0.0 & 0.0 \\
-10.0 & 0.0 & 0.0 & -10.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-25.0 & 0.0 & 0.0 & -25.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
-25.0 & 0.0 & 0.0 & -25.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -10.0 & 0.0 & 0.0 & -10.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -25.0 & 0.0 & 0.0 & -25.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & -25.0 & 0.0 & 0.0 & -25.0 & 0.0 & 0.0
\end{bmatrix}
$$

The column vector q gives the immediate gain of using this policy, each element of the column vector q is given by:

$$
q_i = \sum_{j=1}^{j=N} p_{ij} r_{ij} \tag{12}
$$

Therefore, the immediate gain of this policy is:

$$
q = \begin{bmatrix}
-10.0 \\
-25.0 \\
-25.0 \\
-10.0 \\
-25.0 \\
-25.0 \\
-10.0 \\
-25.0 \\
-25.0
\end{bmatrix}
$$

Note that the state transition probability values for each of the 27 policies can be derived and the respective immediate gains calculated. These values shall be used in policy improvement and value-determination iteration.

Optimal Policy Generator

1. *Value-Determination Operation*

   Use p_{ij} and q_i for a given policy vector to solve

   $$g + v_i = q_i + \sum_{j=1}^{N} p_{ij} v_j \qquad i = 1, 2, \ldots, N \tag{13}$$

   for all relative values v_i and g by setting v_N to zero.

2. *Policy-Improvement Routine*

   For each state i, find the policy k' that maximizes

   $$q_i^k + \sum_{j=1}^{N} p_{ij}^k v_j \tag{14}$$

   using the relative values v_i of the previous policy (see [How60]). Then k' becomes the new policy in the ith state, $q_i^{k'}$ becomes q_i, and $p_{ij}^{k'}$ becomes p_{ij}.

Few notes on the Optimal Policy Generator:

1. The above procedure works only when the discrete markov process is completely ergodic. That is, any discrete markov process whose limiting probability distribution is independent of the starting conditions. Also an assumption is made that all policies produce completely ergodic processes. Complete ergodicity means that each policy defines a discrete Markov process with one recurrent chain. The above algorithm can be extended to generate optimal policies when there are more than one recurrent chains. The details are given in [How60]. All these algorithms have been implemented.

2. The algorithm works by either starting at step 1 or step 2 and iterating over them till successive iterations do not change the optimal policy. The variable g gives the gain of using a policy vector. The gain is monotonically increasing with each iteration. The gain and relative values v_i for each state can be calculated by solving the simultaneous linear equations 13. The relative values are interpreted as giving importance to each of the states with respect to all other states. The relative values hold the key to finding optimal policy vector.

With respect to our example $N = 9$, the number of states in the discrete Markov process, the number of alternative policies are 27. Since we are starting with an initial policy vector $Policy$ the set of equations described in step 1 can be defined and solved. This gives the gain with this policy as -13.33. The values are given by v column vector as:

$$v = \begin{bmatrix} 26.66 \\ 16.66 \\ 11.66 \\ 23.33 \\ 13.33 \\ 8.33 \\ 15.00 \\ 5.00 \\ 0.0 \end{bmatrix}$$

These values with the q values are used to generate the new policy vector as defined in step 2. This policy vector p after the first iteration is:

$$Policy = \begin{bmatrix} 1 \\ 1 \\ 3 \\ 1 \\ 4 \\ 1 \\ 1 \\ 4 \\ 3 \end{bmatrix}$$

This policy vector gives rise to a new stochastic matrix S and new reward matrix R and new immediate benefit vector q. The set of equations specified in step 1 can again be derived and solved to give relative values and gain. Thus this process takes place four more times before converging on an optimal policy vector given by:

$$Policy = \begin{bmatrix} 3 \\ 2 \\ 3 \\ 4 \\ 4 \\ 4 \\ 3 \\ 5 \\ 6 \end{bmatrix}$$

The algorithms for the value-determination and policy iteration have been implemented and were used to generate this optimal policy vector.

Appendix C

SELECT STATEMENT DESCRIBING DATA ACCESSED BY TRANSACTIONS

Each transaction that is initiated by an application accesses some tuples and columns of the relations. Therefore, the part of the relation accessed by each of the transaction initiated by the applications can be described by SQL SELECT statements. Below are the list of transactions initiated application-wise, the data accessed by them, and the site at which they are initiated.

Application P1. accesses data defined by the following SQL statements initiated by its transactions:

At Site CoC

T1. SELECT Recr–id, Recr–name, Recr–add, Recr–type
FROM Recruiters
WHERE Recr–college = 'CoC';

T2. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–college = 'CoC' and Recr–type = 'ug';

T3. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–college = 'CoC' and Recr–type = 'grad';

At Site CoS

T4. SELECT Recr–id, Recr–name, Recr– add, Recr–type
FROM Recruiters
WHERE Recr–college = 'CoS';

T5. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–college = 'CoS' and Recr–type = 'ug';

T6. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–college = 'CoS' and Recr–type = 'grad';

At Site CoE

T7. SELECT Recr–id, Recr–name, Recr– add, Recr–type
FROM Recruiters
WHERE Recr–college = 'CoE';

T8. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–college = 'CoE' and Recr–type = 'ug';

T9. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–college = 'CoE' and Recr–type = 'grad';

At Site CoM

T10. SELECT Recr–id, Recr–name, Recr–add, Recr–type
FROM Recruiters
WHERE Recr–college = 'CoM';

T11. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–college = 'CoM' and Recr–type = 'ug';

T12. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–college = 'CoM' and Recr–type = 'grad';

At Site CAB

T13. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–type = 'ug';

T14. SELECT Recr–id, Recr–name, Recr–add
FROM Recruiters
WHERE Recr–type = 'grad';

Application P2. accesses data defined by the following SQL statements initiated by its transactions:

At Site CoC

T15. SELECT Pros–id, Pros–name, Pros–add, Pros–sex, Pros–ms, Pros–dob
FROM Prospects
WHERE Pros–college = 'CoC';

T16. SELECT Pros–id, Pros–resi, Pros–app–id, Pros–term–yr
FROM Prospects
WHERE Pros–college = 'CoC';

T17. SELECT Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–college = 'CoC' and Pros–type = 'ug';

T18. SELECT Pros–id, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM Prospects
WHERE Pros–college = 'CoC' and Pros–type = 'grad';

At Site CoS

T19. SELECT Pros–id, Pros–name, Pros–add, Pros–sex, Pros–ms, Pros–dob
FROM Prospects
WHERE Pros–college = 'CoS';

T20. SELECT Pros–id, Pros–resi, Pros–app–id, Pros–term–yr
FROM Prospects
WHERE Pros–college = 'CoS';

T21. SELECT Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–college = 'CoS' and Pros–type = 'ug';

T22. SELECT Pros–id, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM Prospects
WHERE Pros–college = 'CoS' and Pros–type = 'grad';

At Site CoE

T23. SELECT Pros–id, Pros–name, Pros–add, Pros–sex, Pros–ms, Pros–dob
FROM Prospects
WHERE Pros–college = 'CoE';

T24. SELECT Pros–id, Pros–resi, Pros–app–id, Pros–term–yr
FROM Prospects
WHERE Pros–college = 'CoE';

T25. SELECT Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–college = ‘CoE’ and Pros–type = ‘ug’;

T26. SELECT Pros–id, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM Prospects
WHERE Pros–college = ‘CoE’ and Pros–type = ‘grad’;

At Site CoM

T27. SELECT Pros–id, Pros–name, Pros–add, Pros–sex, Pros–ms, Pros–dob
FROM Prospects
WHERE Pros–college = ‘CoM’;

T28. SELECT Pros–id, Pros–resi, Pros–app–id, Pros–term–yr
FROM Prospects
WHERE Pros–college = ‘CoM’;

T29. SELECT Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–college = ‘CoM’ and Pros–type = ‘ug’;

T30. SELECT Pros–id, Pros–gpa, Pros–gmat
FROM Prospects
WHERE Pros–college = ‘CoM’ and Pros–type = ‘grad’;

At Site CAB

T31. SELECT Pros–id, Pros–name, Pros–add, Pros–sex, Pros–ms, Pros–dob
FROM Prospects
Pros–state = ‘GA’ and Pros–gpa > 3.0;

T32. SELECT Pros–id, Pros–name, Pros–add, Pros–sex, Pros–ms, Pros–dob
FROM Prospects
WHERE Pros–gpa > 3.25 and Pros–state in {South East}

T33. SELECT Pros–id, Pros–name, Pros–add, Pros–sex, Pros–ms, Pros–dob
FROM Prospects
WHERE Pros–international = 1 and Pros–toefl > 600;

Application P3. accesses data defined by the following SQL statements initiated by its transactions:

At Site CAB

T34. SELECT Event–name, Event–place, Event–st–date, Event–end–date
FROM Events
WHERE Event–type = 'ug';

T35. SELECT Event–name, Event–place, Event–st–date, Event–end–date
FROM Events
WHERE Event–type = 'grad';

Application NP1. accesses data defined by the following SQL statements initiated by its transactions:

At Site CoC

T36. SELECT Pros–id, Recr–id
FROM Prospects, Recruiters
WHERE Pros–type = Recr–type and Recr–college = 'CoC';

At Site CoS

T37a. SELECT Pros–id, Recr–id
FROM Prospects, Recruiters
WHERE Pros–type = Recr–type and Recr–college = 'CoS';

At Site CoE

T37. SELECT Pros–id, Recr–id
FROM Prospects, Recruiters
WHERE Pros–type = Recr–type and Recr–college = 'CoE';

At Site CoM

T38. SELECT Pros–id, Recr–id
FROM Prospects, Recruiters
WHERE Pros–type = Recr–type and Recr–college = 'CoM';

Application NP2. accesses data defined by the following SQL statements initiated by its transactions:

At Site CAB

T39. SELECT Recr–id, Pros–id, Pros–name, Pros–add

FROM Prospects, ProsRecr
WHERE Adv–st–date > Todays–date - 1 and Pros–id = Recr–id;

Application NP3. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T40. SELECT Pros–id, Pros–name, Pros–sex, Pros–ms, Pros–state, Pros–add, Pros–resi, Pros–app–id, Pros–term–yr
FROM Prospects;

At Site CoC

T41. SELECT Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–college = 'CoC' and Pros–type = 'ug';

T42. SELECT Pros–id, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal, Pros–toefl
FROM Prospects
WHERE Pros–college = 'CoC' and Pros–type = 'grad';

At Site CoS

T43. SELECT Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–college = 'CoS' and Pros–type = 'ug';

T44. SELECT Pros–id, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal, Pros–toefl
FROM Prospects
WHERE Pros–college = 'CoS' and Pros–type = 'grad';

At Site CoE

T45. SELECT Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–college = 'CoE' and Pros–type = 'ug';

T46. SELECT Pros–id, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal, Pros–toefl
FROM Prospects

WHERE Pros–college = ‘CoE’ and Pros–type = ‘grad’;

At Site CoM

T47. SELECT Pros–id, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–college = ‘CoM’ and Pros–type = ‘ug’;

T48. SELECT Pros–id, Pros–gpa, Pros–gmat
FROM Prospects
WHERE Pros–college = ‘CoM’ and Pros–type = ‘grad’;

Application NP4. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T49. SELECT *
FROM Recruiters;

Application NP5. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T50. SELECT Recr–id, Pros–id, Pros–name, Pros–add
FROM Prospects, ProsRecr
WHERE Pros–dob = (Todays–date + 7) and Pros–id = Recr–id;

Application NP6. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T51. SELECT Recr–id, Recr–name, Pros–id, Pros–name, Event–id, Event–name,
Event–place, Pros–add, Recr–add
FROM Recruiters, Prospects, Events
WHERE Event–place = Pros–place and Pros–college = Recr–college
and Pros–type = Recr–type and Event–type = Pros–type;

Application NP7. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T52. SELECT Pros–id, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–type = 'ug';

Application NP7. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T53. SELECT Pros–id, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM Prospects
WHERE Pros–type = 'grad';

Application NP9. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T54. SELECT Pros–id, Pros–toefl
FROM Prospects
WHERE Pros–international = 1;

Application NP10. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T55. SELECT Pros–id, Pros–gmat
FROM Prospects
WHERE Pros–type = 'grad' and Pros–college = 'CoM';

Application NP11. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T56. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–type = 'ug';

T56a. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM Prospects
WHERE Pros–type = 'grad';

T57. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–gmat
FROM Prospects
WHERE Pros–type = 'grad';

At Site CoC

T58. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–type = 'ug' and Pros–college = 'CoC';

T59. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM Prospects
WHERE Pros–type = 'grad' and Pros–college = 'CoC';

At site CoS

T60. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–type = 'ug' and Pros–college = 'CoS';

T61. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM Prospects
WHERE Pros–type = 'grad' and Pros–college = 'CoS';

At site CoE

T62. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects
WHERE Pros–type = 'ug' and Pros–college = 'CoE';

T63. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM Prospects
WHERE Pros–type = 'grad' and Pros–college = 'CoE';

At site CoM

T64. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM Prospects

WHERE Pros–type = 'ug' and Pros–college = 'CoM';

T65. SELECT Pros–id, Pros–sex, Pros–gpa, Pros–gmat
FROM Prospects
WHERE Pros–type = 'grad' and Pros–college = 'CoM';

Application NP12. accesses data defined by the following SQL statements initiated by its transactions:

At Site CBA

T66. SELECT Recr–id, Recr–name, Recr–type, Recr–college, Pros–id, Pros–sex,
Pros–gpa, Pros–sat–verb, Pros–sat–quan, Pros–sat–anal
FROM ProsRecr, Prospects, Recruiters
WHERE Prospects.Pros–id = ProsRecr.Pros–id and
ProsRecr.Recr–id = Recruiters.Recr–id and
Pros–type = 'ug';

T67. SELECT Recr–id, Recr–name, Recr–type, Recr–college, Pros–id, Pros–sex,
Pros–gpa, Pros–gre–verb, Pros–gre–quan, Pros–gre–anal
FROM ProsRecr, Prospects, Recruiters
WHERE Prospects.Pros–id = ProsRecr.Pros–id and
ProsRecr.Recr–id = Recruiters.Recr–id and
Pros–type = 'grad';

T68. SELECT Recr–id, Recr–name, Recr–type, Recr–college, Pros–id, Pros–sex,
Pros–gpa, Pros–gmat
FROM ProsRecr, Prospects, Recruiters
WHERE Prospects.Pros–id = ProsRecr.Pros–id and
ProsRecr.Recr–id = Recruiters.Recr–id and
Pros–type = 'grad' and Pros–college = 'CoM';

BIBLIOGRAPHY

[ACL87] R. K. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, December 1987.

[Ape88] P. M. G. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems*, 13(3):263–304, September 1988.

[BCN91] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin-Cummins, Inc., 1991.

[CABK88] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1988.

[Cas72] R. G. Casey. Allocation of copies of a file in an information network. In *Proceedings of Spring Joint Computer Conference, IFIPS,*, pages 617–625, 1972.

[CDY90] B. Ciciani, D. M. Dias, and P.S. Yu. Analysis of replication in distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):247–261, June 1990.

[Chu69] W. W. Chu. Optimal file allocation in a multiple computer system. *IEEE Transactions on Computers*, C-18(10), 1969.

[Chu92] P-C. Chu. A transaction-oriented approach to attribute partitioning. *Information Systems*, 17(4):329–342, 1992.

[CI91] W. W. Chu and I. T. Ieong. A transaction-based approach to vertical partitioning for relational databases. Technical report, UCLA, 1991.

[CL88] M. J. Carey and M. Livny. Distributed concurrency control performance: A study of algorithms, distribution, and replication. In *Proceedings of International Conference on Very Large Databases*, 1988.

[CMP80] S. Ceri, G. Martella, and G. Pelagatti. Optimal file allocation for a distributed on a network of minicomputers. In *Proceedings of International Conference on Databases, Aberdeen*, pages 345–357, July 1980.

[CNP82] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 128–136, 1982.

[CNW83] S. Ceri, S. B. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Transactions on Software Engineering*, SE-9(5):487–504, July 1983.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw Hill, New York, 1984.

[CPW88] S. Ceri, B. Pernici, and G. Wiederhold. Distributed database design methodologies and tools. In *Proceedings of the IEEE*, pages 533–546, May 1988.

[CY87] D. W. Cornell and P. S. Yu. A vertical partitioning algorithm for relational databases. In *Proceedings of International Conference on Data Engineering*, pages 30–35, February 1987.

[DGG+86] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *Proceedings of International Conference on Very Large Databases*, 1986.

[DYB87] D. M. Dias, P.S. Yu, and B. T. Bennett. On centralized versus geographically distributed database systems. In *Proceedings of International Conference on Distributed Computing Systems*, September 1987.

[Esw74] K. P. Eswaran. Placement of records in a file and file allocation in a computer network. *Information Processing*, pages 304–307, 1974.

[Goe93] G. Goetz. Volcano, an extensible and parallel dataflow query processing system. *IEEE Transactions on Knowledge and Data Engineering*, 1993.

[GP82] B. Gavish and H. Pirkul. Allocation of databases in distributed computing systems. *Management of Distributed Data Processing*, pages 215–231, 1982.

[GP86] B. Gavish and H. Pirkul. Computer and database location in distributed computer systems. *IEEE Transactions on Computers*, C-35(7):583–590, 1986.

[How60] R. A. Howard. *Dynamic Programming and Markov Processes.* M.I.T. Press, 1960.

[HS75] J. A. Hoffer and D. G. Severance. The use of cluster analysis in physical database design. In *Proceedings of International Conference on Very Large Databases*, pages 69–86, 1975.

[MCVN93] J. Muthuraj, S. Chakravarthy, R. Varadarajan, and S. B. Navathe. A formal approach to the vertical partitioning problem in distributed database design. In *Proc. of Second International Conference on Parallel and Distributed Information Systems, San Diego, California*, 1993.

[MSW72] W. T. McCormick, P. J. Schweitzer, and T. W. White. Problem decomposition and data reorganization by a clustering technique. *Operation Research*, 20(5):993–1009, September 1972.

[NCWD84] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9(4):680–710, 1984.

[NR89] S. B. Navathe and M. Ra. Vertical partitioning for database design: A graphical algorithm. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1989.

[NRJ90] D. M. Nicol and P. F. Reynolds Jr. Optimal dynamic remapping of data parallel computations. *IEEE Transactions on Computers*, Febrruary 1990.

[Ra90] Minyoung Ra. *Data Fragmentation and Allocation Algorithms for Distributed Database Design*. PhD thesis, University of Florida, 1990.

[RVVN90] P. I. Rivera-Vega, R. Varadarajan, and S. B. Navathe. Scheduling data redistribution in distributed databases. In *Proceedings of International Conference on Data Engineering*, February 1990.

[RW83] C. V. Ramamoorthy and B. W. Wah. The isomorphism of simple file allocation. *IEEE Transactions on Computers*, C-32(3):221–232, March 1983.

[SI91] D. Shin and K. B. Irani. Fragmenting relations horizontally using a knowledge-based approach. *IEEE Transactions on Software Engineering*, 17(9), September 1991.

[SKL89] K. G. Shin, C. M. Krishna, and Y. H. Lee. Optimal dynamic control of resources in a distributed system. *IEEE Transactions on Software Engineering*, October 1989.

[TGS85] Y. C. Tay, N. Goodman, and R. Suri. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, December 1985.

[TSG85] Y. C. Tay, R. Suri, and N. Goodman. A mean value performance model for locking in databases: The no-waiting case. *Journal of the ACM*, 32(3):618–651, July 1985.

[VRVN89] R. Varadarajan, P. I. Rivera-Vega, and S. B. Navathe. Data redistribution scheduling in fully connected networks. In *Proceedings of 27th Annual Allerton Conference on Communication, Control and Computing*, September 1989.

[WN86] B. Wilson and S. B. Navathe. An analytical framework for the redesign of distributed databases. In *Proceedings of the 6th Adavanced Database Symposium, Tokyo, Japan.*, 1986.

[YDCI87] P. S. Yu, D. M. Dias, D. W. Cornell, and B. R. Iyer. Analysis of affinity based routing in multi-system data sharing. *Performance Evaluation*, 7(2):87–109, June 1987.

[YDR+85] P. S. Yu, D. M. Dias, J. T. Robinson, B. R. Iyer, and D. W. Cornell. Modelling of centralized concurrency control in multi-system environment. In *Proceedings of ACM SIGMETRICS*, pages 183–191, 1985.

[YDR+87] P. S. Yu, D. M. Dias, J. T. Robinson, B. R. Iyer, and D. W. Cornell. On coupling multi-systems through data sharing. In *Proceedings of the IEEE*, pages 573–587, May 1987.

[YSLS85] C. T. Yu, C. Suen, K. Lam, and M. K. Siu. Adaptive record clustering. *ACM Transactions on Database Systems*, 10(2):180–204, June 1985.