

9501

AN INTELLIGENT INTERFACE FOR A DISTRIBUTED DATABASE

TC YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

by
İlknur ŞANSLI

February, 2000
İZMİR

AN INTELLIGENT INTERFACE FOR A DISTRIBUTED DATABASE

**A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of
Dokuz Eylül University
In Partial Fulfillment of the Requirements for
the Degree of Master of Science in Computer Engineering,
Computer Engineering Program**

by

İlknur ŞANSLI

February, 2000

İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Doç. Dr. Alp Kut
(Advisor)

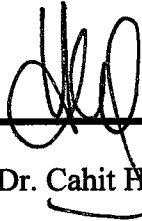


Doç. Dr. Özgür Dikenelli
(Committee Member)



Doç. Dr. Yılmaz Güler
(Committee Member)

Approved by the
Graduate School of Natural and Applied Sciences



Prof. Dr. Cahit Helvacı
Director

ACKNOWLEDGEMENTS

In choosing the topic “Distributed Database Management Systems” for this study, the wish to learn Distributed Systems and my interest in Database Management Systems have a big role. Honestly, this was a long and difficult research, because of the wide range the topic includes, and the lack of formal definitions for true distributed database systems. Nevertheless, with the wish to complete this study and under the auspices of my adviser Doç. Dr. Alp Kut, my fiancé, and my family, it has been possible to overcome all the difficulties. I want to thank them all for their support.

İlknur ŞANSLI

ABSTRACT

The aim of this thesis is investigating the topic “Distributed Database Systems” and implementing a prototype, namely “An Intelligent Interface for a Distributed Database”. The concept “Distributed Database System” has a wide range including almost all the topics discussed in the computer science literature and needing a big coordination of all these topics which can be gathered together under the combined title of two base topics : Database Management and Networking. As a result of this wide range, the prototype discussed is implemented on a subrange of the topic.

Nowadays, in which we are at the beginning of year 2000; we are aware of the fact that, from now on, no two points should be unaware of each other whatever their distance may be. This is the real need and driving force for a distributed database technology; because of the changing customer demands and market needs, business operations became more decentralized geographically. On the other hand, we still need a central and global view of our systems. Then what we need is the system that allows distribution of data, but also provides opportunities to integrate them. Here the most important feature of a distributed database system comes on the scene, which is called transparency. Due to this feature, users of the distributed database system are allowed to manage a physically dispersed database as if it were a centralized database.

When we mention physically dispersed data, heterogeneity is unavoidable. According to our needs, we may have different limits on the heterogeneity, and these limits will contribute to different design alternatives. The overall system can be homogeneous, meaning that each site will have the same database management system; or they can be heterogeneous having different database management

systems. Here, another important concept in the design of a distributed database is autonomy, determining the degree of independence for each database management system, participating in the distributed database. These concepts are important when we decide to design and use a distributed database system, and there are some important topics that we should never forget: Fragmentation, Allocation and Replication. These topics are interested in availability and reliability of the distributed database, distribution of processing load, and storage cost reduction to provide high performance, reliability and functionality. Since we have decentralized business needs, we have to distribute our data. This implies to partition our data to manageable units, called fragmentation. We want to distribute the data over different sites to provide higher availability by locating it near the greatest demand site. This process, in other words optimal distribution of the fragments to existing sites is called allocation. In this distributed environment, we should also be able to access our data even in the case of site failures. This need could be satisfied by replicating the same data unit or fragment into several sites called replication. Accordingly, fragmentation, allocation and replication are three important design concepts.

Briefly, distributed database management system brings the advantages of increased reliability and availability, local control over data, modular growth, lower communication costs and faster response, in return for a price of management and control complexity. Distributed database technology is one of the most important developments of recent times, and it has been the subject of intense research and development effort. Now, we are in the critical point of the transition to commercial products. At the beginning I mentioned that the distributed database topic includes a fairly wide range of topics. Today, all these topics are well researched. However, as the integrated problem that covers all these areas is NP-hard, much of the existing work has either concentrated on only one of these problems or restricted the problem space. Accordingly, this thesis has the aim of examining all of these topics independently, and implementing a prototype in a restricted problem space.

ÖZET

Bu tezin amacı Dağıtık Veri Tabanı Sistemleri konusunda araştırma yapmak ve dağıtık veri tabanı üzerinde çalışan akıllı bir arayüz prototipi geliştirmektir. Dağıtık Veri Tabanı Sistemi kavramı bilgisayar bilimleri literatüründe yer alan konuların hemen hemen hepsini içeren geniş bir alana sahiptir ve bütün bu konuların koordinasyonunu gerektirir. Bu konular Veri Tabanı Yönetimi ve Ağ Teknolojileri ana başlıkları altında toplanabilir. Konunun bu kadar geniş bir alanı kapsaması nedeniyle, prototip, konunun daraltılmış bir kısmı üzerine gerçekleştirilmiştir.

2000 yılının başlangıcında olduğumuz günümüzde, bizler bundan böyle aralarındaki uzaklık ne olursa olsun, birbirinden habersiz olan iki nokta olmayacağı gerçeğinin bilincindeyiz. Bu gerçek dağıtık veri tabanı teknolojisinin ortaya çıkmasındaki en önemli etken ve itici güçtür. Değişen müşteri ve pazar ihtiyaçları günümüzde, coğrafi açıdan merkezi olmayan bir yapı gerektirmektedir. Öte yandan, her ne kadar dağıtık olsa da bizler hala sistemlerimizin merkezi ve global görünümüne ihtiyaç duymaktayız. Bu durumda ihtiyacımız olan sistem, verilerimizi dağıtmaya izin verdiği gibi, bunları bir araya getirmek için gerekli olanakları da bizlere sunmalıdır. Bu noktada, dağıtık veri tabanı sisteminin en önemli özelliği olan saydamlık ortaya çıkmaktadır. Bu özellik sayesinde, dağıtık veri tabanı kullanıcıları fiziksel olarak dağıtılmış bir veri tabanını merkezi bir veri tabanı imiş gibi yönetme olanağına kavuşur.

Fiziksel olarak dağıtılmış verilerden söz ettiğimizde, heterojenlik kaçınılmazdır. İhtiyaçlarımıza göre, heterojenlik konusunda farklı limitlerimiz olabilir. Bu limitler de farklı tasarım alternatiflerini ortaya çıkaracaktır. Tüm sistem homojen olabilir,

yani sistem içerisinde yer alan her bölge aynı veri tabanı yönetim sistemini kullanabilir; ya da bölgeler heterojen olabilir, dolayısıyla her bir bölge farklı veri tabanı yönetim sistemine sahip olabilir. Burada dağıtık veri tabanı tasarımında önemli olan bir diğer kavram ortaya çıkmaktadır: Özerklik. Bu kavram dağıtık veri tabanında yer alan her bir veri tabanının bağımsızlık derecesini saptar. Bu kavramlar dağıtık veri tabanı tasarımına ve kullanımına karar verildiğinde üzerinde önemle durulması gereken kavramlardır. Aynı zamanda unutulmaması gereken üç önemli konu daha vardır. Bunlar verilerin parçalanması, atanması ve kopyalanması işlemleridir. Bu konular dağıtık veri tabanının kullanılabilirliği, yüksek performans amacı ile işlem yükünün dağıtımı ve veri depolama maliyetinin azaltılması, veri tabanının güvenilirliği ve fonksiyonelliği ile ilgilidir. Coğrafi açıdan dağıtık olan iş ihtiyaçlarımız nedeniyle verilerimizi dağıtmaya gerek duyduğumuzu belirtmiştik. Bu dağıtım parçalama adı verilen, verilerimizin yönetilebilir parçalara bölünmesi işini gerektirmektedir. Bölünmeden elde edilen her bir parçaya erişimi en uygun hale getirmek için, her bir parça en çok ihtiyaç duyulduğu bölgeye yerleştirilmelidir. Var olan parçaların bulunan bölgelere optimal dağıtımı işine atama denir. Bu dağıtık ortamda elbette verilere, çeşitli aksaklıklardan dolayı belirli bölgelere erişim engellendiğinde bile ulaşma gereksinimi doğacaktır. Bu gereksinim de kopyalama adı verilen belirli veri gruplarının ya da parçalarının birden fazla bölgeye kopyalanması işlemiyle karşılanabilir. Bütün bu bilgiler ışığında, parçalama, atama ve kopyalama üç önemli tasarım kavramıdır.

Özetle, her ne kadar yönetimi ve kontrolü merkezi sistemlere göre daha güç olsa da, dağıtık veri tabanı yönetim sistemi, sistemin güvenilirliğini ve kullanılabilirliğini artırır, veri üzerinde yerel kontrol sağlar, sistemin modüler büyümesini kolaylaştırır, iletişim maliyetini azaltır ve veriye daha hızlı erişime olanak tanır. Dağıtık veri tabanı teknolojisi, son yılların en önemli gelişmelerindendir ve bu konuda günümüze dek yoğun bir araştırma ve gelişme elde edilmiştir. Artık yapılan tüm araştırmaların ticari uygulamalara geçirilmesi gibi kritik bir noktada bulunmaktadır. Başlangıçta da değindiğimiz gibi, dağıtık veri tabanı konusu çok geniş bir alanı kapsayarak birçok konuyu içinde barındırmaktadır. Bu kapsama giren tüm konular üzerinde ayrı ayrı kapsamlı araştırmalar yapılmıştır, ancak bütün bu konuların entegrasyonu

bişimşel bir problemdir. Bu nedenle mevcut arařtırmalar ya kapsam içindeki konulardan sadece birine konsantre olmuş, ya da problemin alanında kısıtlamalar yapmışlardır. Bütün bunlara baęlı olarak, bu tezin amacı, daęıtık veri tabanı konusunun kapsamındaki bütün bu konuları incelemek ve belirli varsayımlara göre kısıtlanmış olan bir prototip geliřtirmektir.



CONTENTS

	Page
Contents	VIII
List of Figures	XI

Chapter One INTRODUCTION

1.1 Summary	1
1.2 Organization of Thesis	5

Chapter Two PRELIMINARY TOPICS FOR DISTRIBUTED DATABASES

2.1 Brief introduction to DBMS	7
2.2 Brief introduction to Distributed Systems	11

Chapter Three

INTRODUCTION TO DISTRIBUTED DATABASES

3.1 What is a Distributed Database?	17
3.2 The need for Distributed Database technology	19
3.3 Distributed Database Design	22
3.3.1 Fragmentation	22
3.3.1.1 Types of fragmentation	24
3.3.1.2 Ways to test the correctness of decomposition	30
3.3.2 Replication	30
3.3.3 Allocation	35
3.4 Important concepts in the design of Distributed Databases	36
3.4.1 Transparency	36
3.4.2 Autonomy	38
3.5 Architectural models for Distributed DBMSs	40
3.6 Query Processing	41
3.7 Transaction Management	45
3.7.1 Introduction to Transactions	47
3.7.2 Concurrency Control Protocols	48
3.7.3 Reliability Protocols	54

Chapter Four

DIFFERENT CLIENT-SERVER DATABASE SYSTEMS

4.1 Client-Server Architecture for Centralized Databases	62
4.2 Client-Server Architecture for Distributed Databases	65

Chapter Five

THE CURRENT STATE AND FUTURE EXPECTATIONS

5.1 The state of the current products.....	67
5.1.1 Oracle	70
5.1.2 Sybase.....	84
5.1.3 PeerDirect.....	90
5.1.4 IBM Solution – DRDA	95
5.1.5 Mariposa Distributed Database Management System.....	97
5.2 Unsolved problems	101
5.3 Future expectations	104

Chapter Six

INTRODUCTION TO AN INTELLIGENT INTERFACE

6.1 The Structure of a Name Server Database	107
6.2 Query Processing	108
6.2.1 Data Retrieval Queries	110
6.2.1.1 Selection from one table	110
6.2.1.2 Selection from multiple tables -Join Operation-.....	117
6.2.2 Data Manipulation (Insert, Update, Delete) Queries.....	120
6.2.3 Pseudo Codes	126
Conclusions.....	133
References	135

LIST OF FIGURES

	Page
Figure 1.1 The layered view on a distributed system.....	3
Figure 1.2 Centralized Client/Server Model	4
Figure 1.3 Distributed Client/Server Model.....	4
Figure 2.1 An example of a table	8
Figure 2.2 Traditional File Processing	10
Figure 2.3 Database Processing	10
Figure 2.4 Computer Network	11
Figure 2.5 Star Network.....	14
Figure 2.6 Hierarchical (Tree) Network.....	15
Figure 2.7 Ring Network.....	15
Figure 2.8 Meshed Network.....	16
Figure 3.1 Distributed Database Environment.....	18
Figure 3.2 Complete Comparison of Fragmented Tables	28
Figure 3.3 Partitioned join	29
Figure 3.4 Simple join graph.....	29
Figure 3.5 Architectural models for distributed DBMSs	41
Figure 3.6 2PL Lock Graph	50
Figure 3.7 Strict 2PL Lock Graph.....	50
Figure 3.8 Communication Structure of Centralized 2PL	51
Figure 3.9 Communication Structure of Distributed 2PL	52
Figure 3.10 Centralized 2PC Communication Structure.....	56
Figure 3.11 State Transitions in 2PC Protocol.....	56
Figure 3.12 State Transitions in 3PC Protocol.....	58

Page

Figure 4.1 SCS Architecture	63
Figure 4.2 CS-MD Architecture.....	63
Figure 4.3 E-CS Architecture.....	65
Figure 4.4 An Example of a Distributed DBMS Architecture.....	66
Figure 5.1 The Client/Server Architecture and Distributed Processing.....	71
Figure 5.2 Two different uses of Primary site Ownership	76
Figure 5.3 Dynamic Ownership	77
Figure 5.4 Shared Ownership.....	79
Figure 5.5 Multiple Master Replication	80
Figure 5.6 Updatable Snapshots.....	81
Figure 5.7 Hybrid Configuration.....	82
Figure 5.8 Deferred RPCs.....	83
Figure 6.1 E ² R Schema of the Name Server Database	107
Figure 6.2 E ² R Schema of an example Student Database.....	108
Figure 6.3 Fragmentation Schema of an example Student Database.....	109
Figure 6.4 Allocation Schema of an example Student Database	110
Figure 6.5 The main role of Fragmenter	111
Figure 6.6 An Interface for Data Retrieval Queries.....	116
Figure 6.7 Selection from a replicated table	117
Figure 6.8 Generic Query.....	118
Figure 6.9 Reduced Query	119
Figure 6.10 Query with a Join Operation.....	120
Figure 6.11 Insertion into the Student table.....	122
Figure 6.12 The result of the insert operation from the user's point of view	123
Figure 6.13 Update Propagation Example	125
Figure 6.14 Queue Mechanism	125

CHAPTER ONE

INTRODUCTION

1.1. Summary

A distributed database can be defined as a database that is not entirely stored at a single physical location, but rather is dispersed over a network of interconnected computers. In other words, a distributed database is under the control of a database management system in which storage devices are not all attached to a common processor. Simply a distributed database is a union of several databases placed in different physical locations.

The distributed database concept is brought to light because of the decentralized business needs. Trying to reach the data being kept in a central database from remote locations suffer problems such as performance degradation and reliability problems created by dependence on a central site. Distributed databases on the other hand, allow partitioning data into manageable parts called fragments and allocating every fragment to the greatest demand sites, therefore trying to maximize local processing and improving communications. In addition, distributed databases eliminate single point of failure possibility, and facilitate growth, by enabling to add new sites without affecting the others. Beside these advantages, it is harder to design and manage a distributed database and to maintain data security.

A Distributed Database Management System must operate in a network environment which consists of computer workstations, network hardware and software, communication media, transaction processors (distributed transaction

managers) and data processors (local transaction managers or local data managers). A distributed database can be designed by using data replication, fragmentation (horizontal partitioning or vertical partitioning) or combinations of replication and fragmentation. There are two design alternatives: top-down and bottom-up design. In a top-down fashion, an existing centralized database is separated into more than one database on different sites, thus forming a logically single distributed database. The need for this change can be performance degradation of the centralized system. On the contrary, in a bottom-up design the system is decentralized and there are existing separate databases, these databases maintain their data locally, and update the central database at regular intervals, meaning that data is not a shared resource. Because these databases are on non-networked computers, each computer has access only to the central site and therefore exchange of data among these sites is too expensive. So, organizations with physically dispersed sites, each maintaining its own data, are candidates for a distributed database. Here the solution would be combining these databases into a single virtual distributed database.

The most important feature of a distributed database is transparency, meaning that the users of a distributed database are not aware of the fact that their operations involve multiple sites and databases. This feature allows users to use a physically dispersed database as if it were a centralized database. To make this illusion, there should be a global schema (a global view of database schema similar to centralized database), fragmentation schema (a schema to partition the database into logical fragments), allocation schema (a schema to determine the allocation of fragments to each site, with or without replication) and local mapping schema (the schema of each independent DBMS). By using the information gathered from these levels, a distributed database management system can manage user requests and users will not be aware of the distribution.

A Distributed Database Management System (DDBMS) must be able to provide additional functions to those of a centralized DBMS. Some of these are:

1. To access remote sites and transmit queries and data among the various sites via a communication network.
2. To keep track of the data distribution and replication in the DDBMS catalog.
3. To devise execution strategies for queries and transactions those access data from more than one site.
4. To decide on which copy of a replicated data item to access.
5. To maintain the consistency of copies of a replicated data item.
6. To maintain the global conceptual schema of the distributed database
7. To recover from individual site crashes and from new types of failures such as failure of a communication link.

It is obvious that a distributed database management system requires a distributed system. The layered view on a distributed system can be seen in Figure 1.1.



Figure 1.1 The layered view on a distributed system.

In a distributed system, client server model is widely used. Traditional client server model has the structure, which can be seen in Figure 1.2.

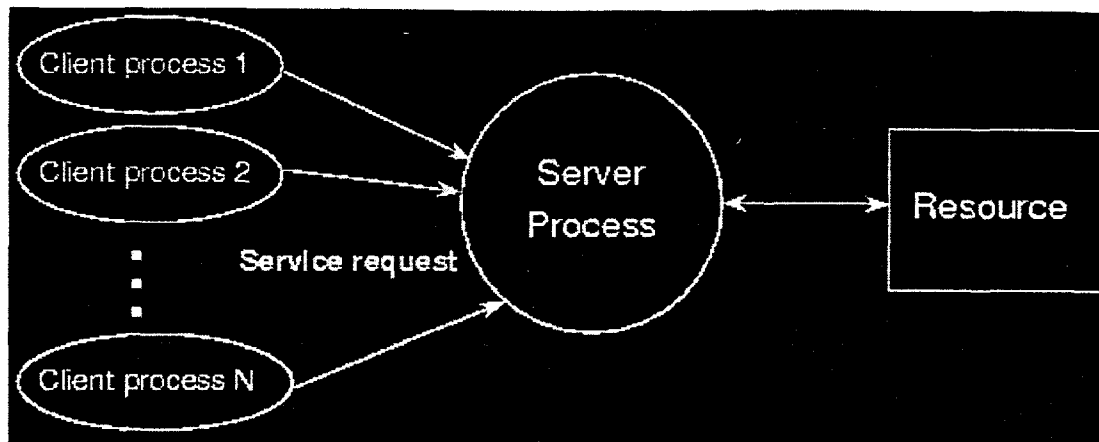


Figure 1.2 Centralized Client/Server Model

There are major problems with this basic client/server model. The control of individual resources is centralized in a single server and each single server is a potential bottleneck. To improve the performance, multiple implementations of similar functions can be used. The improved model can be seen in Figure 1.3.

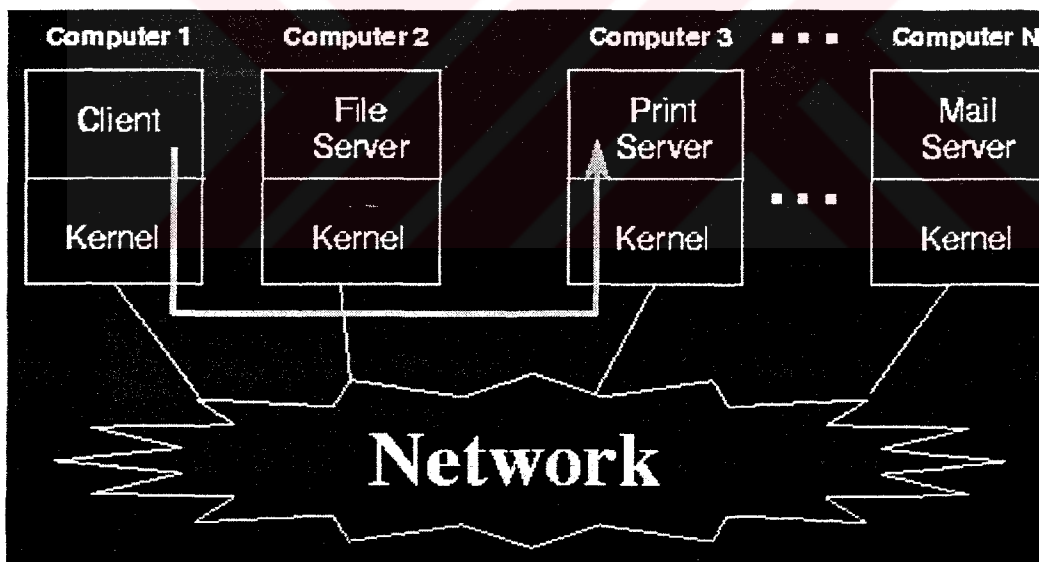


Figure 1.3 Distributed Client/Server Model

This model is also used for distributed database systems. Suppose that each computer in Figure 1.3 has a DBMS running on it. Therefore all these computers become database servers and one server may need to access a database on another

server. In this case, the server requesting the information becomes a client.

In [COT5200], rules for distributed systems are given. These are:

0. To the user, a distributed system should look exactly like a nondistributed system
1. Local autonomy
2. No reliance on a central site
3. Continuous operation
4. Location independence
5. Fragmentation independence
6. Replication independence
7. Distributed query processing
8. Distributed transaction management
9. Hardware independence
10. Operating system independence
11. Network independence
12. DBMS independence

Different levels in the distribution, autonomy, and heterogeneity forms different architectural models of distributed databases. These models can be seen in Chapter 3, Figure 3.5.

1.2. Organization of Thesis

Chapter two has the aim of making an entrance to the topic “Distributed Databases” from the starting point. This starting point has two main titles, DBMS (Database Management System) and Distributed Systems. In section 2.1, data and data model concepts are introduced and the need for DBMS is explained. In section 2.2, distributed system and basic networking concepts are introduced.

In chapter three, “distributed database” topic is explained in depth. First distributed database is defined and the need for distributed databases is explained. Then some important concepts like transparency and autonomy, design issues like fragmentation, replication and allocation, architectural models, query processing and transaction management topics are examined in detail.

In chapter four, client-server model is adopted to distributed databases and client-server database systems are introduced.

Chapter five makes an evaluation of today and the future. In section 5.1, several current products are introduced and their approaches to the topics of section four are explained. In section 5.2, unsolved problems are mentioned and, in section 5.3, future expectations are discussed.

Chapter six is an introduction to an intelligent interface. In this chapter, after explaining assumptions made in the implementation of the interface, in section 6.1, the structure of the Name Server Database is given. In 6.2, the way in which an interface processes data retrieval and data manipulation queries is introduced.

CHAPTER TWO

PRELIMINARY TOPICS FOR DISTRIBUTED DATABASES

2.1. Brief introduction to DBMS

This section has an objective of giving an idea about what data is, how it can be managed and under these considerations about data management, what is the promise of a DBMS.

What does “data” mean?

We can define data as the information that we use and need to store for further access. Facts and figures we use or store are data [Ozkarahan, 1997]. Data is represented in several formats including numbers, text strings, images, and voice. When data stored in files on disks or other media has no meaning, it is only physical. But when the interpretation of data is made according to the enterprise being the purpose of that data, the logical structure of data will be formed. Logical structure of data is the way that data is understood by its users.

What does “data model” mean?

We don't need to store the data if it doesn't have any meaning for us. If it has meaning, some of the data will contribute to the same purpose and will be used together by having some relationship among them. This relationship is the logical

structure of the data and has a model. As a result, data model is the shape of the relationship among data which is determined according to the enterprise.

What kind of data models do we have?

A database is a structured collection of data related to some real-life phenomena that we are trying to model [Ozsu & Valduriez, 1991]. There are some different kinds of data models including the relational model, the hierarchical model, the network data model and the semantic role model. In fact, these models are our point of views to data. I want to mention here the relational model which is the most famous one.

In a relational database, we have different data sets having some relationships among them. These datasets form a relation. For example, assume we have a set of NAMES (these names have different meanings for every enterprise, i.e. if we are interested in a school, these names may belong to the students or lecturers, if our interest is a company, the names will belong to the employees of the company) and a set of NUMBERS (according to their types, these numbers can be the school_numbers of the students or salaries of the employees). If the names and numbers have meaning, each student will have a school_number, or each employee will have a salary. Assume our enterprise is a school. I can structure the relationship between the NAMES and NUMBERS as a table shown in Figure 2.1.

STUDENT	
Student_number	Student_name

Figure 2.1 An example of a table

Each row in this table will belong to a student. Each cell under the Student_number column is from the domain NUMBERS and the cells under the Student_name column is from the domain NAMES. This table in the relational model is called as a relation.

“Formally, a relation R defined over n sets D_1, D_2, \dots, D_n (not necessarily distinct) is a set of n -tuples (or simply tuples) $\langle d_1, d_2, \dots, d_n \rangle$ such that $d_1 \in D_1, d_2 \in D_2, \dots, d_n \in D_n$ ” [Ozsu & Valduriez, 1991, p.18]. Thus, in a relational model, relations are two-dimensional tables whose elements are data items. This is a very simple structure and allows a high degree of independence from the physical data representation.

The relational model provides a solid foundation to data consistency. Consistent states of a database can be uniformly defined and maintained through integrity rules. The relational model also allows set-oriented manipulation of relations. This feature has led to the development of powerful nonprocedural languages based either on set theory (relational algebra) or on logic (relational calculus). These are the most powerful features of relational model [Ozsu & Valduriez, 1991].

What is a DBMS?

A Database Management System (DBMS) is the result of an important need called “data independence”. It can be defined as the separation of the implementation details of application programs and manipulation of data which is necessary for application programs. By this way, applications are independent from the changes of physical manipulation of data and accordingly, physical changes on data will not require any change in applications. With the development of database management systems, instead of data processing in which each application defines and maintains its own data, the idea of defining and administering data centrally is adopted (Figure 2.2 and Figure 2.3 from [Ozsu & Valduriez, 1991]). Thus, an application programmer is responsible for only the details of application and let DBMS take care of data manipulation.

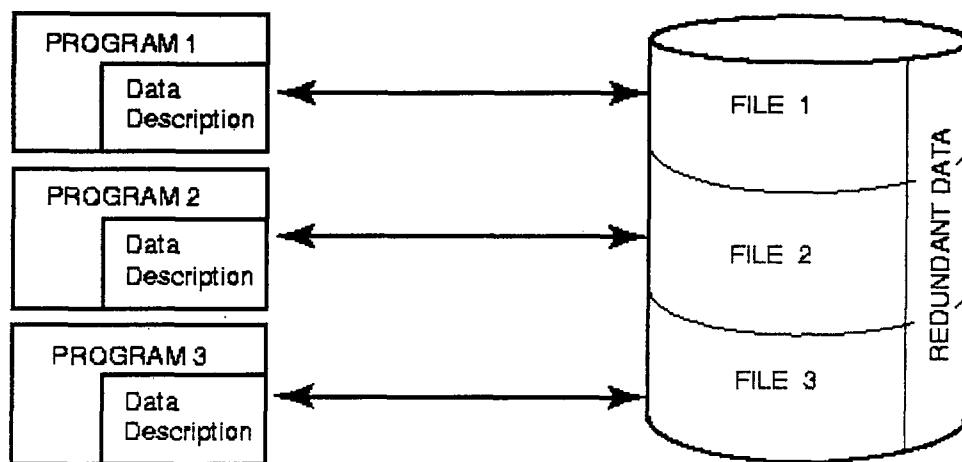


Figure 2.2 Traditional File Processing

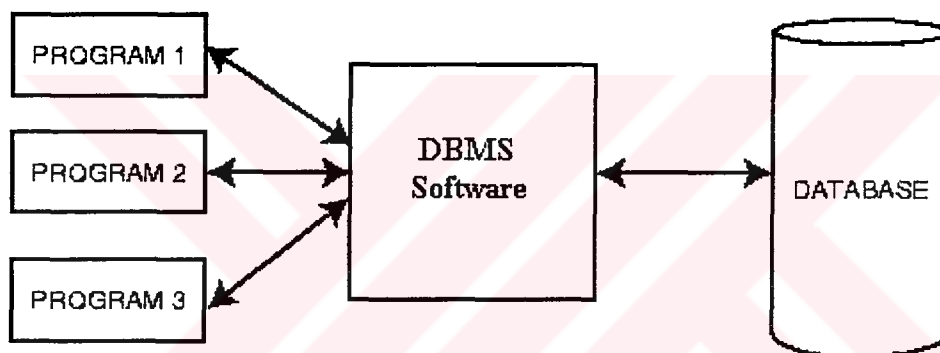


Figure 2.3 Database Processing

In [Ozkarahan, 1997] DBMS is defined as a software system embodying

- a data model
- a data language for definition and manipulation of data
- means to enforce and implement security, integrity, and concurrency.

2.2. Brief introduction to Distributed Systems

The goal of this section is to provide the idea of distribution. Thus, it will be clear that what is the need for distribution, what kind of things may be distributed, and how they can be interconnected?

What is a distributed system?

A distributed system is a collection of independent equipments interconnected by a communication medium to do a common work. In this system, if we have computers as equipments, we will have this definition of [Tanenbaum, 1995, p.2]: “A distributed system is a collection of independent computers that appear to the users of the system as a single computer.” In this environment, the communication among independent computers is procured by different network topologies (communication channels) and the system can also be called as a computer network. “Computer network is an interconnected collection of autonomous computers that are capable of exchanging information among them” [Ozsu & Valduriez, 1991, p.42]. In Figure 2.4 general structure of a computer network can be seen.

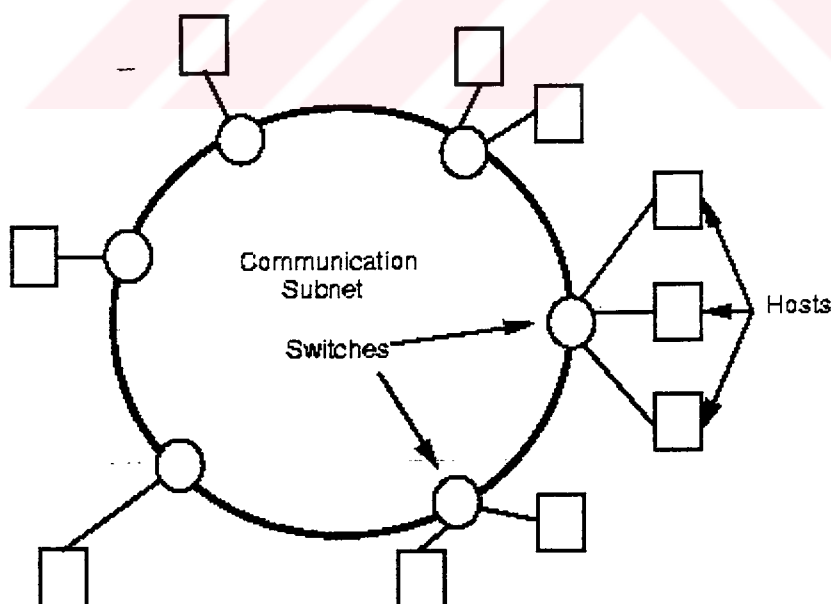


Figure 2.4 Computer Network

The Need for Distribution

With the development of powerful microprocessors and high-speed networks, it is understood that distributed systems have a much better price/performance ratio than a single large centralized system. In addition to this real driving force, the existence of inherently distributed applications need interconnection among different sites which are autonomous in some degree. Distributed systems also eliminate single point of failure possibility which is the most dissuasive disadvantage of a central system; thus distribution is more reliable. In a distribute system, incremental growth is made simply by adding more processors. Briefly, we can say that, the main reason for distribution is the need for people to work together and share information in a convenient way in spite of the fact that they are in different places.

What could be distributed?

Things that might be distributed are [Ozsu & Valduriez, 1991]:

- Processing logic: Processing logic or processing elements could be distributed.
- Function: Various functions of a computer system could be delegated to various pieces of hardware or software.
- Data: Data used by a number of applications may be distributed to a number of processing sites.
- Control: The control of the execution of various tasks might be distributed instead of being performed by one computer system.

In distributed database systems, all of these distribution modes are important.

Basic Networking Concepts

In a computer network, individual computers are communicating with each other by the help of protocols. “A protocol is an agreement between the communicating parties on how communication is to proceed” [Tanenbaum, 1995, p.35]. There are

two general types of protocols : **connection oriented** and **connectionless**. In the connection oriented protocols, before exchanging data, a connection is established between the sender and receiver. When data exchanging is done completely, the connection is terminated. In the connectionless protocols, no setup for communication is needed. The sender transmits the first message when it is ready. There may be different types of environments that is suitable for either connection oriented or connectionless protocols. Connection oriented protocols can cause a considerable overhead because of the connection establishment phase, although they are more reliable. Thus the choice is depending on our expectations.

A protocol between the communicating parties might be very complex, because agreements can be needed at a variety of levels, varying from the low-level details of bit transmission to the high-level details of how information is to be expressed. To simplify the task of a protocol, it can be structured in a layered way, with each layer having the responsibility of different agreement details. Open Systems Interconnection Reference Model (OSI) developed by International Standards Organization (ISO) is a standard model for layered protocols. The OSI model is designed to allow open systems to communicate. In the OSI model, communication is divided into seven layers. Each layer deals with one specific aspect of the communication that must be agreed on by the communicating sites. OSI is a connection oriented protocol. Its layers are:

1. Application layer- user interface layer.
2. Presentation layer- data representation layer.
3. Session layer - enables two applications to communicate across a network.
4. Transport layer - assures reliable transmission.
5. Network layer - sets up pathways.
6. DataLink layer - puts messages together with headers.
7. Physical layer - transmits bits over the physical medium.

Each layer has its own protocol that can be changed independently of the other one. This characteristic of the layered protocols makes it more attractive than the others.

But when the overhead of connection oriented protocols is considered, connectionless protocols can be preferred. The Client/Server model is based on a simple, connectionless request/reply protocol. In this model, there are two types of processes. The process that requests a service is called a client, and the process that replies to the request coming from client is called a server. This is also a layered protocol, but here the number of layers are decreased, because here no connections are established, and therefore no need for session management. The client sends a request to the server asking for some service. The server processes the work and returns the requested data or an error indicating the problem.

Computer networks can be classified according to various criteria. These are [Ozsu & Valduriez, 1991]:

- The interconnection structure of computers (topology): As the name indicates, interconnection structure or topology refers to the way computers on a network are interconnected. Below, some different topologies can be seen. _

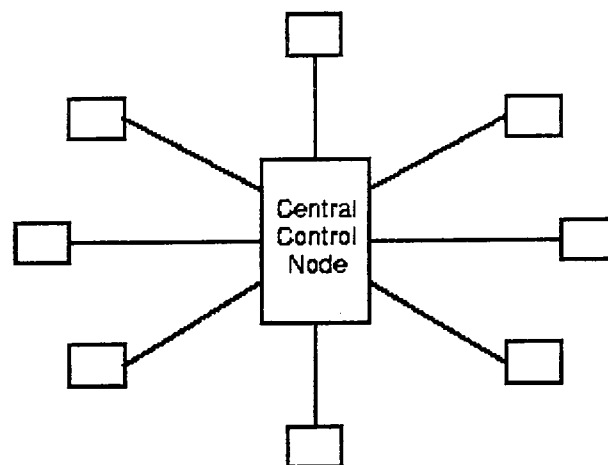


Figure 2.5 Star Network

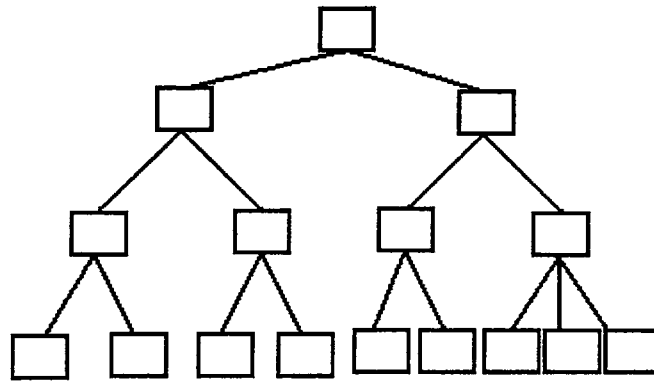


Figure 2.6 Hierarchical (Tree) Network

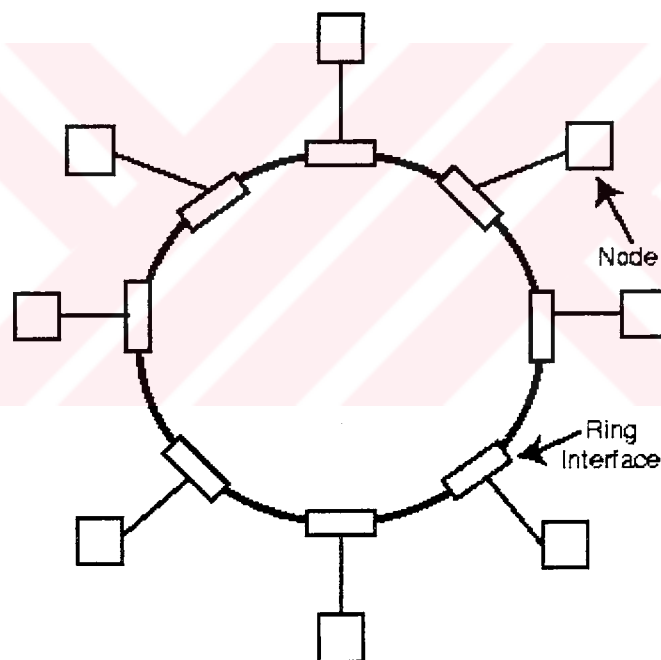


Figure 2.7 Ring Network

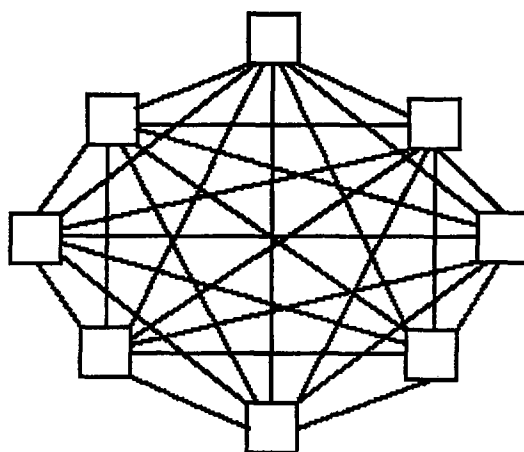


Figure 2.8 Meshed Network

Besides these topologies above, in the irregular topology, the interconnection between nodes does not follow a pattern.

- Communication schemes.
 1. Point-to-point networks: In point to point networks each pair of nodes has a link connecting them. This link may not be a direct link, it may be an indirect link with intermediate connections.
 2. Multi-point (Broadcast) networks: In multi-point networks, a common communication channel is utilized by all the nodes in the network.
- Geographic distribution: This classification is made according to geographic scope, in other words the distance between any two nodes of the network.
 1. WAN (Wide Area Network).
 2. LAN (Local Area Network).

CHAPTER THREE

INTRODUCTION TO DISTRIBUTED DATABASES

3.1. What is a Distributed Database?

A Distributed Database is a single logical database that is spread physically across computers in multiple locations that are connected by a data communications link. In [Ozsu & Valduriez, 1991], it is defined as a collection of multiple, logically interrelated databases distributed over a computer network. Here the distinction between the first and the second definitions is important to understand the concept. In the first definition, distributed database is defined as a **single logical database** that is spread across multiple locations, although it is defined as **multiple, logically interrelated databases** in the second one. An important point is that the collection of multiple databases are making up a logical single database in spite of the fact that they are located at different nodes of the network.

Each part of the distributed database is under the control of a separate DBMS running on an independent computer system. Each system has autonomous processing capability serving local applications. Each system participates, as well, in the execution of one or more global applications. Such applications require data from more than one site. This requirement implies the existence of a software that manages these separate DBMSs and integrates them to preserve the logical integration. This software is a distributed database management system (DDBMS). In [Ozsu & Valduriez, 1991], DDBMS is defined as the software system that permits

the management of the distributed database (DDB) and makes the distribution transparent to the users. Distributed Database Environment can be seen in Figure 3.1.

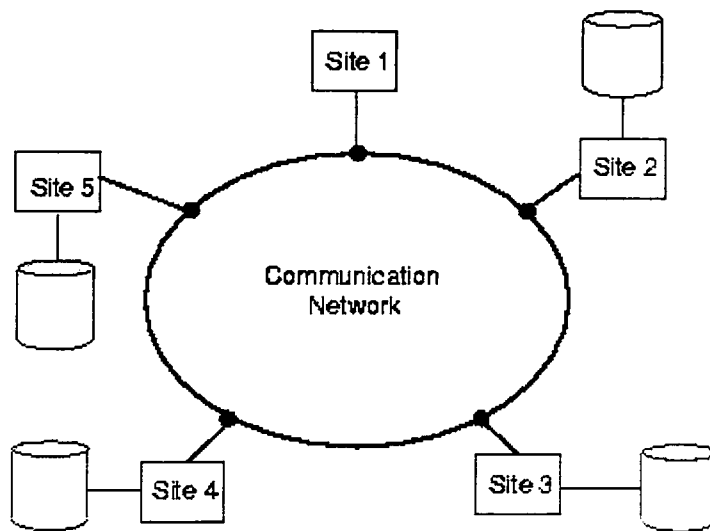


Figure 3.1 Distributed Database Environment

According to the definitions given above, these assumptions, which are valid in today's technology base, can be made [Casavant & Singhal, 1994]:

1. Data is stored at a number of sites. Each site is assumed to logically consist of a single processor. Even if some sites are multiprocessor machines, the distributed DBMS is not concerned with the storage and management of data on this parallel machine.
2. The processors at these sites are interconnected by a computer network rather than a multiprocessor configuration. The important point here is the emphasis on loose-interconnection between processors that have their own operating systems and operate independently.
3. The DDB is a database, not some "collection" of files that can be individually stored at each node of a computer network. This is the distinction between a DDB and a collection of files managed by a distributed file system. To form a DDB, distributed data should be logically related (where the relationship is defined according to some structural formalism) and access to data should be at

a high level (via a common interface). The typical formalism that is used for establishing the logical relationship is the relational model. In fact, most existing distributed database system research assumes a relational system.

3.2. The Need for Distributed Database technology

Most business networks fall into one of two categories. The first category includes organizations using a large centralized database. In this environment, users rely solely on IS (Information System) departments to provide needed information. The second category includes organizations where each user or department has its own collection of data. The organization uses hard copy or replication to share that data. In this category, by using decentralized database, the network overhead for accessing the database is eliminated but it is not possible having a global view of the system. Thus, neither strategy satisfies the need to respond to market changes quickly and cost-effectively in today's competitive business environment. This inefficiency of two common types of business solutions (centralized, decentralized) is the real driving force for a distributed database. In [Richter, 1994], distributed database is accepted as a solution with these reasons:

By allowing data to be accessed locally and managed globally, a fully functional distributed database can supply reliable information anytime and anywhere. It provides users timely and flexible access to information, it gives them the tools to analyze their data in more meaningful ways, it optimizes the use of computer processor power, and it lets the IS department control the safety and integrity of the data.

Centralized systems store data in tables in a central DBMS. Users access the tables, which are on more powerful machines (e.g., a Unix-based minicomputer or a mainframe, which has a high costed database system), concurrently through either dumb terminals or their PCs via a LAN. This configuration still solves many of the problems with maintaining data integrity, eliminating redundant data, and processing changes, deletions, and updates concurrently. Centralized databases also simplify

routine services (e.g., security, back up, and maintenance). Nevertheless, when dealing with remote access, centralized systems are at the mercy of the communications lines resulting in performance degradation. Also, they generally require a dedicated staff--at least a database administrator, and often an entire IS department--to construct, optimize, and maintain the database. Another important disadvantage of centralized systems is their reliability problems created by dependence on a central site, which is a single point of failure. When organizations with a centralized database has outgrown the capacity --in terms of either storage or processing power-- of their current database server or they begin to suffer from the performance degradation of their centralized system, this situation requires separating a logically centralized database into a distributed database that spans two or more computing processes. Typically, this is two computers sitting side by side (but it may include geographically separated sites) running compatible DBMS products. Many of the now available commercial database servers support horizontal scaling, which lets database systems include additional servers or processors. The database system transparently distributes the data and the database processing load. Most top-down operations in the business world today work with a comparably sized or scaled-down version of the original centralized database. This approach ensures transparency while maintaining data integrity and applications compatibility. Designing a distributed database by this way is called as a **top-down design** approach.

We can define a decentralized database as a collection of independent databases on non-networked computers. A decentralized configuration has both data and applications located on independent sites. Key to the decentralized configuration is the concept that the data is not a shared resource. Each site maintains its data locally and updates the central database at regular intervals. These sites can share data, but they typically lack the facilities (e.g., mechanisms and procedures for communication and data-integrity controls) to do so. Exchange of data among these sites is often difficult and expensive. So, organizations with physically dispersed sites, each maintaining its own data, are candidates for a distributed database. A possible solution for this type of organization might be to combine all the dispersed databases

into one central database. However, we just mentioned several disadvantages of central databases. Instead of a centralized database, a bottom-up integration of a distributed database --combining existing databases running on mixed systems into a single, virtual distributed database-- solves these problems. This approach preserves an organization's investment in database software and applications, as well as allowing the data to be stored where it's used most. Designing a distributed database by this way is called as a **bottom-up design** approach.

In situations where the data needs to be a shared resource, a distributed multiuser database may be the solution. As with the decentralized model, each site has an independent CPU and DBMS, as well as its own data and applications. Also, each site has an added component to enable shared data: a TM (transaction manager). The TM analyzes a request for data from the user and directs the request to the appropriate server. That server acts upon the request as if issuing it locally (thus distributing the processing) and returns the answer set to the requesting TM. The requesting TM analyzes, collates, and stores the replies from each server, and eventually the user sees the result.

We can arrange the needs for distributed databases briefly as:

- Business operations became more decentralized geographically and local business units want control over data.
- Competition increased at the global level.
- Customer demands and market needs favoured a decentralized management style and there is a need for consolidation of data across local databases for integrated decision making.
- Microcomputers increased in power, which encourage the growth of local area networks, as mainframe costs are high.
- DDBMS encourages data sharing
- DDBMS reduces telecommunication costs.
- DDBMS reduces the risk of telecommunication failures.

As a result, the distributed database technology intends to extend the concept of data independence to environments where data are distributed and replicated over a number of machines connected by a network.

3.3. Distributed Database Design

In the design of distributed database systems, there are four major aspects that must be considered [March & Rho, 1995]:

1. The communications network (location of nodes, allocation of computer resources, network topology, and selection of link capacities),
2. Data allocation (determining the units of data to allocate -fragmentation- and allocating copies of those units to nodes),
3. Operating strategies related to query optimization and concurrency control (determining which copy or copies of data to access, where to process the data, how to route the data, locking and commit protocols),
4. Local database design (record structures, record placement algorithms, secondary indexes, processing algorithms).

3.3.1. Fragmentation

In a distributed DBMS to partition the database without regard of physical location of data, is called “fragmentation”. A relational table may be broken up into two or more non-overlapping partitions or slices. A table may be broken up horizontally, vertically, or a combination of both. Partitions may in turn be replicated. The reason for this partitioning is because the relation is not a suitable distribution unit.

In a distributed environment, it is desirable to perform as much tasks as possible at the local level without any access to the other sites. This is an important performance issue, because the local processing provides an easier management and a more efficient execution. Although a complete locality is an aim from the

performance point of view, it can not be realized due to the distribution requirements of the system. When the logic of applications is considered, it can be seen that every application is interested in different parts of a relation. While some applications are interested in only some tuples of relations thus needing the parts of the relations obtained by horizontal partitioning, others are interested in only some attributes of relations obtained by vertical partitioning. Since the locality of accesses of applications is defined not on entire relations but on their subsets, it is only natural to consider subsets of relations as distribution units.

When different applications residing at different sites have views on some relation and an entire relation is considered as a distribution unit, there are two alternatives. First alternative is storing a relation at only one site without replication. In this case, there will be an unnecessarily high volume of remote data accesses. Second alternative is replicating a relation on all or some of the sites where the application resides. This case, on the other hand, has unnecessary replication, which causes problems in executing updates and may not be desirable if storage is limited. Thus, fragments of relations are the most convenient distribution units for distributed systems.

The decomposition of a relation into fragments, each being treated as a unit, permits a number of transactions to execute concurrently. In addition, the fragmentation of relations typically results in the parallel execution of a single query by dividing it into a set of subqueries that operate on fragments. Thus fragmentation typically increases the level of concurrency and therefore the system throughput. Besides the advantages of fragmentation especially in local access optimization, there are also some disadvantages of fragmentation. If the applications have conflicting requirements which prevent decomposition of the relation into mutually exclusive fragments, those applications whose views are defined on more than one fragment may suffer performance degradation. Because in such situations, it is necessary accessing data across partitions. It might, for example, be necessary to retrieve data from two fragments and then take either their union or their join, which is costly. Avoiding this is a fundamental fragmentation issue.

The extent to which the database should be fragmented is an important decision that affects the performance of query execution. The measurement of fragmentation degree is called as granularity. The degree of fragmentation goes from one extreme, that is not to fragment at all, to the other extreme, to fragment to the level of individual tuples (in the case of horizontal fragmentation) or to the level of individual attributes (in the case of vertical fragmentation) [Ozsu & Valduriez, 1991]. If it is too low, then it needs a lot of management cost. If it is too rough (user level) then the unnecessary elements should be replicated causing a higher cost. In a vertical fragmentation, combining data across partitions is more difficult because it requires joins. What we need, then, is to find a suitable level of fragmentation, which is a compromise between the two extremes. Such a level can only be defined with respect to the applications that will run on the database.

3.3.1.1. Types of Fragmentation

There are two fundamental fragmentation strategies: horizontal and vertical. Furthermore, there is a possibility of nesting fragments in a hybrid fashion (mixed fragmentation). It is possible to see that many real-life partitioning may be hybrid.

Horizontal Fragmentation: In this type, the relation is partitioned horizontally into fragments obtained as a selection operation on the global relation. Granularity is at the tuple level, and the attribute values of the tuple determine the corresponding fragment meaning that different records (tuples) of a relation will be at different sites. Thus each fragment has a subset of the tuples of the relation.

Below, there are some basic definitions from [Ozsu & Valduriez, 1991, pp.106, 107], which will be helpful to understand how a relation is divided into subrelations:

Given a relation $R(A_1, A_2, \dots, A_n)$, where A_i is an attribute defined over domain D_i , a simple predicate p_j defined on R has the form $P_j : A_i \theta \text{Value}$ where $\theta \in \{=, <, \neq, <=, >, >=\}$ and Value is chosen from the domain of A_i ($\text{Value} \in D_i$). We use Pr_i to denote the set of all simple predicates defined on a relation R_i . The members of Pr_i

are denoted by p_{ij} . Even though simple predicates are quite elegant to deal with, user queries quite often include more complicated predicates, which are Boolean combinations of simple predicates. One combination that we are particularly interested in, called a minterm predicate, is the conjunction of simple predicates. Given a set $Pr_i = \{p_{i1}, p_{i2}, \dots, p_{im}\}$ of simple predicates for relation R_i , the set of minterm predicates $M_i = \{m_{i1}, m_{i2}, \dots, m_{iz}\}$ is defined as $M_i = \{m_{ij} \mid m_{ij} = \bigwedge p_{ik}^*, \text{ where } p_{ik} \in Pr_i \text{ and } 1 \leq k \leq m, 1 \leq j \leq z\}$ where $p_{ik}^* = p_{ik}$ or $p_{ik}^* = \neg p_{ik}$. So each simple predicate can occur in a minterm predicate either in its natural form or its negated form.

According to the definition of a minterm predicate given above, a horizontal fragment can be defined as the set of tuples for which a minterm predicate is true. Thus there are as many horizontal fragments of relation R as there are minterm predicates. Here, the way in which predicates and minterm predicates are obtained is an important design issue. From the above definitions of [Ozsu & Valduriez, 1991], we know that, the definition of horizontal fragments depends on minterm predicates and minterm predicates are obtained from simple predicates. This means that first we have to find simple predicates. If fragmentation is the need of distributed applications, it is obvious that design of fragmentation will be made according to application needs. So, we have to analyze all user applications and find the predicates used in user queries. If it is not possible to analyze all of the user applications to determine these predicates, we should at least investigate the most “important” ones. By this way we will obtain the groups of tuples having different references by at least one application and find a set of simple predicates for every fragment groups. A set of predicates can be used to describe the fragmentation if they are complete and minimal.

A set of simple predicates P_r is said to be complete if and only if there is an equal probability of access by every application to any two tuples belonging to any minterm fragment that is defined according to P_r . In other words, P is *complete* if and only if any two tuples belonging to the same fragment are referenced with the same probability by any application. Fragments obtained this way are not only logically

uniform in that they all satisfy the minterm predicate, but statistically homogeneous. To obtain a complete set of predicates automatically require the designer to specify the access probabilities for each tuple of a relation for each application under consideration.

The second desirable property of the set of predicates which is minimality simply states that if a predicate influences how fragmentation is performed (i.e., causes a fragment f to be further fragmented into, say, f_i and f_j), there should be at least one application that accesses f_i and f_j differently. In other words, the simple predicate should be relevant in determining a fragmentation. If all the predicates of a set P_r are relevant, P_r is minimal. A predicate is *relevant* if it contributes to the distinction of fragments with different access patterns.

After determining the set of simple predicates, which are both complete and minimal, we can derive minterm predicates on them. These minterm predicates determine the fragments that are used as candidates in the allocation step. In the determination of individual minterm predicates, there is a difficulty of encountering quite large set of minterm predicates. Thus we should eliminate some of the minterm fragments that may be meaningless. Elimination is performed by finding minterms that might be contradictory to a set of implications I . This example from [Ozsu & Valduriez, 1991, p.113] will help to better understand the process of finding simple predicates, accordingly finding minterm predicates and elimination of minterm predicates.

For example, if $P_r' = \{p_1, p_2\}$, where $p_1 : att = value_1$; $p_2 : att = value_2$ and the domain of att is $\{value_1, value_2\}$, it is obvious that I contains two implications, which state

$$i_1 : (att = value_1) \Rightarrow \neg(att=value_2)$$

$$i_2 : \neg(att = value_1) \Rightarrow (att=value_2)$$

The following four minterm predicates are defined according to P_r' :

$$m_1 : (att = value_1) \wedge (att=value_2)$$

$$m_2 : (att = value_1) \wedge \neg(att=value_2)$$

$$m_3 : \neg(att=value_1) \wedge (att=value_2)$$

$$m_4 : \neg(att=value_1) \wedge \neg(att=value_2)$$

In this case the minterm predicates m_1 and m_4 are contradictory to the implications I and can therefore be eliminated from M .

There are two types of horizontal fragmentation, **primary horizontal fragmentation** and **derived horizontal fragmentation**. Before defining these types separately, I want to define the terms source (owner) and target (member) relations being related with the join connection between relations. ‘The relation at the tail of a link is called the owner of the link and the relation at the head is called the member’ [Ozsu & Valduriez, 1991].

A primary horizontal fragmentation is defined by a selection operation on the owner relations of a database schema. The formula used in the selection operation is a predicate that is defined on the attributes of the owner relation. Primary horizontal fragments of a relation R can be defined formally as $R_i = \sigma_{F_i}(R)$, $1 \leq i \leq w$ where F_i is the selection formula used to obtain fragment R_i .

In derived horizontal fragmentation on the other hand, fragmentation is defined on a member relation of a link according to selection operation specified on its owner. In other words, member relation is partitioned according to the predicates being defined on its owner relation. The link between the owner and the member relations is defined as an equi-join. The fragmentation of the member relation is made by using semijoin, because we want to partition a member relation according to the fragmentation of its owner, but we also want the resulting fragment to be defined only on the attributes of the member relation. Assume that the owner relation S is fragmented as $S_i = \sigma_{F_i}(S)$ in which each fragment of S is obtained by the formula F_i (here F_i is the selection formula defined on the attributes of S , if F_i is in conjunctive normal form, it is a minterm predicate). To apply the derived horizontal fragmentation to R which is the member relation of S , we should make a semijoin operation between R and each fragment of S . In formal terms, $R_i = R \bowtie S_i$, $1 \leq i \leq w$ where w is the maximum number of fragments that will be defined on R . Then any

fragmentation algorithm will need three inputs which are the set of fragments of the owner relation, the member relation and semijoin predicates between the owner and the member.

Here, one question may be asked: In a database schema, generally there are more than two links into a relation R . In this case there is more than one possible derived horizontal fragmentation of R . Then how should we choose the owner relation according to which derived horizontal fragmentation is made? The decision is dependent on the application characteristics.

In both the primary and the derived horizontal fragmentation, reconstruction of a global relation from its fragments is performed by the union operator. Thus, for a relation R with fragmentation $F_R = \{R_1, R_2, \dots, R_w\}$, $R = \cup R_i, \forall R_i \in F_R$.

After all these information about horizontal fragmentation, it is important to state the way of join operation between fragmented tables. There are several ways to do that. Every tuple of every fragment of a table can be compared with every tuple of every fragment of another table. The join graph can be seen in Figure 3.2.

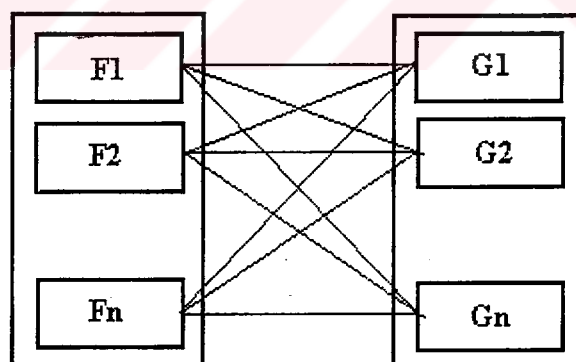


Figure 3.2 Complete Comparison of Fragmented Tables

Although it seems completely true from the operational point of view, it requires high network traffic and a lot of comparisons. The cost can be reduced if not every fragments are to be compared with each other. This way is called as a partitioned join (Figure 3.3).

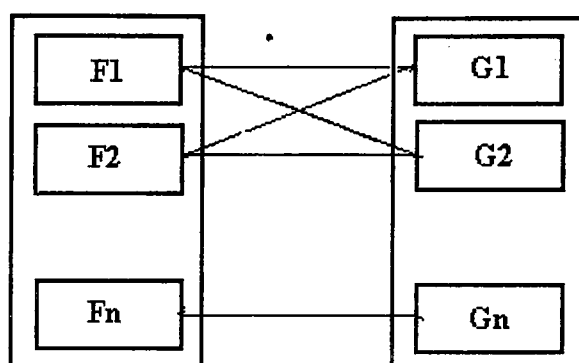


Figure 3.3 Partitioned join

The most optimal case is when every fragment is compared with only one other fragment. Simple join graph (Figure 3.4):

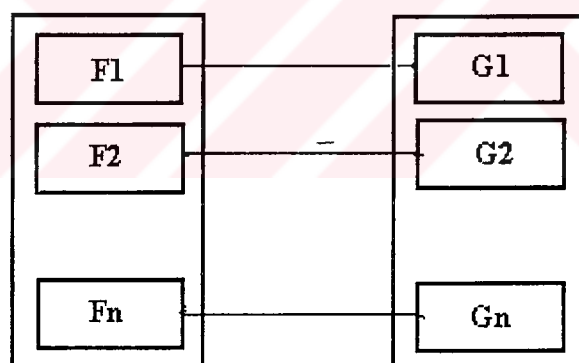


Figure 3.4 Simple join graph

Vertical Fragmentation: In vertical fragmentation different columns (attributes) of a file are distributed to different sites. In this case fragmentation is made by using projection operation on a relation. To preserve unique access, each fragment includes a primary key column. Beside the primary key, according to the application needs, fragmentation is continued by choosing other necessary attributes for each fragment.

In vertical fragmentation, reconstruction of global relations is made by the join operation. Detailed information about this type can be found in [Ozsu & Valduriez, 1991], [Muthuraj et al., 1993].

Mixed Fragmentation: The mixed fragmentation means that a fragment is recursively fragmented, meaning that horizontal fragment might be vertically fragmented or vertically fragment might be horizontally fragmented. Reconstruction can be obtained by applying the construction rules in inverse order. Detailed information about this type can be found in [Ozsu & Valduriez, 1991].

3.3.1.2. Ways to test the correctness of decomposition

There are some rules, which are enforced during fragmentation. These rules ensure semantic consistency of the database during fragmentation. These are **completeness**, **reconstruction** and **disjointness**. Completeness means that any data item that can be found in global relation can also be found in one or more of its fragments. By this way, data in a global relation is mapped into fragments without any loss. As the name implies, reconstruction means composing the global relation from its fragments by using an operator which will be different for every type of fragmentation (union for horizontal, join for vertical fragmentation). The last one is disjointness which has the meaning of the fact that any data item can be found in only one of the fragments of a global relation, in other words, the same data item can not exist in more than one fragment. In vertical partitioning, disjointness is defined only on the nonprimary key attributes of a relation, because its primary key attributes are typically repeated in all its fragments.

3.3.2. Replication

In a distributed DBMS a relational table or a fragment of it may be replicated or copied and copies may be distributed throughout the database. Replication is a very important design issue in distributed databases with several advantages. First, by replication even if some nodes are down, queries can be processed on another copy

of data. Second, read-only queries on the same data item can be executed in parallel, because concurrent accesses to the same data item can be directed to another site which has the replica of that data; by this way fast response will be given to the user queries and work load of any site which has replicated data will be decreased. Thus, replication increase processing power and reliability of the system.

The above advantages of replication are needs of distributed systems that force replication. But, we should consider the difficulties also in applying replication technology. Because data is replicated at more than one place, this will require additional storage space. Replication causes problems especially in update queries, because when some data item is updated, all of the replicas of that data item should be updated and this will bring additional time, cost and complexity to update operations. Besides these, if replicated data is not updated simultaneously, there is a possibility of getting incorrect data. If replicated data has become inconsistent, because the replicated data is updated at multiple sites within the same replication interval, this situation is called an update conflict. Briefly, replication can cause problems for propagating updates and concurrency control. Therefore, the decision regarding replication is a trade-off which depends on the ratio of the read-only queries to the update queries.

Update propagation algorithms are responsible for updates on replicated data. According to the needs of application domain, dimension of time is a very important issue and this question arises: when replicated data is updated, do the updates need to be applied to other copies in real-time or can the updates be propagated on a deferred basis [Oracle 7]? The answer to this question determines the choice of update propagation algorithms. In some systems, temporary inconsistencies in replicated data could be tolerated in a limited time interval, or in others the existence of inconsistency may cause big problems, no matter if this consistency exists for too short or long. According to these application domains, there are two main types of technology. Real-time remote data access and real-time application of updates to replicated data is provided by **synchronous distributed database technology**.

Deferred remote data access and deferred propagation of replicated data updates is provided by **asynchronous distributed database technology**.

With synchronous replication all copies of data are kept exactly synchronized and consistent. If any copy is updated the update will be immediately applied to all other copies within the same transaction. Synchronous replication is appropriate when this exact consistency is important to the business application. Two-phase commit protocol which will be examined in more detail in section 3.7.3 is an algorithm that ensures keeping all replicated data synchronous by adopting the “all or nothing approach”. While two phase commit may be appropriate for situations where a corporation absolutely needs to synchronize distributed data, it comes at a price: Since all distributed sites need to synchronously approve a transaction before it is accepted, if any one site is unavailable, the transaction will have to wait. Operations are therefore exposed to individual component outages. Furthermore, the elaborate handshaking mechanism, with messages going back and forth between sites as they coordinate the acceptance of the data, puts a significant burden on corporate networks. Briefly, all-or-nothing approach is too costly to employ, when the number of sites is very large. In [Shirota et al., 1998], this situation is explained as ‘With eager replication, which updates all replicas whenever a transaction updates any instance of the object, a ten-fold increase in the number of sites would result in a thousand-fold increase in failed transactions (deadlocks)’ and as an alternative to this all-or-nothing approach, a weak consistency model (allows a certain level of asynchronosity and do not require replica updates to proceed in at all sites) called ECHO is introduced.

With asynchronous replication, copies or replicates of data will become temporarily inconsistent with each other. If one copy is updated, the change will be propagated and applied to the other copies as a second step, within separate transactions, that may occur seconds, minutes, hours, or even days later. Copies therefore can be temporarily inconsistent, but over time the data should converge to the same values at all sites. Ensuring convergence in asynchronous replication environments is critical for nearly every application. To ensure convergence, update

conflicts must be detected and resolved, so that the data element has the same value at every site. Alternatively, update conflicts may be avoided by limiting "ownership", or the right to update a given data element, to a single site.

Understanding the tradeoffs between these two technologies is key to determining which to use to solve a particular business problem. The tradeoffs between these two involve application integrity, complexity, performance, and availability and both technologies will be needed to solve different aspects of the problem. Synchronous technology ensures application integrity and minimizes complexity, but can be less available if the systems and networks involved are not reliable. It can also incur poor response time if network access between systems is slow. Asynchronous technology maximizes availability and response time, but can be more complex and requires careful planning and design to ensure application integrity. In asynchronous technology, transactions operate against local data only. They may initiate deferred operations which need to be propagated to other systems but if these other systems are not available the propagation will be deferred until the systems come back up.

Some products use snapshots of changes being one of the implementation methods of asynchronous replication. The snapshot function monitors source table updates and at specified intervals (or on user request) extracts and ships that data to the target or targets to synchronize them. This can be highly efficient, optionally shipping only the net effect of the period's processing, but suffers from the problem of extended periods of inconsistency between the source and target replicas. Snapshots therefore cannot be used to provide continuous or near real time consistency, severely limiting the potential applications. Snapshots also rely on only one of the replicas being the source. Therefore all target replicas must be read only, again limiting the potential applications. Snapshot data replication products are table consistent. That is, they assure that at the completion of a snapshot replication, all copies of a table are consistent. As individual transactions may update several tables, snapshots must be carefully synchronized to prevent inconsistent relationships between tables. Briefly, snapshots do not maintain the integrity of transactions and copy individual data tables or data items without maintaining the atomicity of a

transaction. With transactions broken, the integrity of the distributed data is threatened.

In addition to snapshots, triggers are another asynchronous mechanism provided by some database vendors for the purpose of one-way data replication. A trigger can be thought as an alarm in the database which is associated with a specific piece of data. When a change is made to the tagged data item, that change 'triggers' an alarm inside the source database. The alarm in turn activates replication-specific code inside the source database which begins the replication process. While offering customers more flexibility than snapshots, early trigger-based replication systems did not overcome the fundamental flaw of snapshot technology: the lack of protection of the transactional integrity of customer data. Trigger limitations include:

- Triggers simply transfer individual data items when they are modified, they do not keep track of transactions.
- Triggers allow one-way replication only. Data on replicate sites is read-only and cannot be modified.
- The execution of triggers within a database imposes a performance overhead to that database.
- Triggers require careful management by database administrators. Someone needs to keep track of all the 'alarms' going off when data is modified.
- The activation of triggers in a database cannot be easily 'rolled back' or undone.

To summarize, in early trigger-based replication systems, it is left entirely to the customer to build applications that kept track of and protected the integrity of transactions [Sybase1].

To solve the transactional integrity problem, vendors begin to offer transaction-based replication with triggers or rules. Instead of letting the customers build their own replication systems using triggers or rules as their assembly tools, triggers are used in-house to build replication products, thus isolating customers from the underlying triggers or rules. With the introduction of processes that grouped data

changes into transactions after they were triggered inside the source database, trigger or rule based replication systems solved the problem of losing the transactional integrity of the data. But this method still has the same problems with non-transaction based replication with triggers except the first and second shortcomings.

3.3.3. Allocation

After partitioning the database into fragments, second step in a distributed database design is allocation which means that distributing these fragments into various sites on the network. In determining the proper sites for fragments, minimal cost and maximal performance would be the aim of designers.

Minimal cost consists of storage cost (no unnecessary replications), data retrieval cost (to try to place the fragment on the site of the application), data update cost (to minimize the replication of the read-write data elements) and communication cost (to find the sites near the application and to try to maximize the local processing). Maximal performance consists of minimizing response time (every application should find a replica of the data on the local or a close site) and maximizing throughput (to allow several concurrent applications). In fact, minimizing cost and maximizing performance are closely related issues. When we are trying to minimize cost, we will be maximizing the performance; the contrary case is also true.

There are two main types of fragment allocation strategies: non-redundant and redundant. In non-redundant allocation strategy every fragment is stored at only one site, there is no replication; in redundant allocation strategy a fragment may be replicated on different sites. The redundant storage may rise the efficiency but it is more complicated to design it. According to these main allocation strategies, there are three types of allocation algorithms, which are best-fit algorithm, all beneficial sites algorithm and additional replication algorithm. Best-fit approach is based on non-redundant strategy and the site, which is determined as the best according to measurements, is selected to allocate data. Therefore the data will be at only that site without replication and the system will lack of the advantages coming from

replication. All beneficial sites algorithm is based on redundant strategy in which all the sites that are determined as beneficial in the result of measurements are selected to allocate the data. The goodness of a site is measured by the benefit of the local read accesses and the costs of remote update accesses. This may cause unnecessary replication of data and therefore may result in performance penalties. The third algorithm is trying to decrease these penalties resulting from best-fit and all beneficial sites algorithms by having an idea between these extremes. While it is redundant, because it wants to benefit from replication; it avoids unnecessary replication by starting from a non-redundant allocation and adding beneficial replications to the system if necessary.

Briefly, a non-replicated database (commonly called a partitioned database) contained fragments that are allocated to sites, and there is only one copy of any fragment on the network. In case of replication, either the database exists in its entirety at each site (fully replicated database), or fragments are distributed to the sites in such a way that copies of a fragment may reside in multiple sites (partially replicated database) [Ozsu & Valduriez, 1991].

3.4. Important concepts in the design of Distributed Databases

3.4.1. Transparency

With the improvement of DBMS, the form of data processing in which data definition and maintenance were embedded in applications is left and by abstracting these functions out of the applications and letting them to the responsibility of DBMS, data independence concept is introduced. Therefore, application programs are not aware of the logical and physical organization of data and vice versa. Distributed database technology has the aim of extending the concept of data independence to environments where data is distributed among different machines connected by a network.

Transparent access to data separates the higher level semantics of the system from lower level implementation issues. Thus, the database users would see a logically integrated, single-image database even though it is physically distributed and they will be able to access the distributed database as if it were a centralized one [Casavant & Singhal, 1994]. Briefly, the distributed nature of the database is hidden from users; neither the users nor the programmers need to know where or how the database stores the data. To the user, operations appear to run against one contiguous database. The system alone manages distribution. With this transparency given by the distributed database management system, users continue using familiar database platforms and products, minimizing the cost of software and hardware purchases and retraining users. A distributed approach also allows users to continue using familiar database applications [Richter, 1994].

When we consider the distributed database environment in detail, we can easily find which futures will be transparent to users. First, different from any DBMS environment, there is a network that needs to be managed and the user should not be interested in operational details of the network. This abstraction of user from network is called network (location or distribution) transparency. Second, in a distributed database environment, relations are partitioned into fragments, but user queries are specified on entire relations. In this situation, distributed DBMS should translate the query specified on entire relations to the several queries specified on the fragments. This is called fragmentation transparency. Another thing that must be hidden from user is the replication of data items. The user should act as if there is a single copy of the data. The system should handle the management of copies, and this feature is called replication transparency.

Although transparency is the goal of a distributed database technology, there are still some disagreements in giving full transparency. In [Casavant & Singhal, 1994], poor manageability, poor modularity and poor message performance are given as reasons for being against full transparency. It is obvious that transparent access to data will bring additional complexity to systems, while simplifying the users progress. But transparency, which is the original goal of distributed database

systems, should not be given up. Here, another question arises which is controversial: Who will be responsible for providing transparency? It can be user applications, operating system or distributed DBMS itself. When user applications are responsible for transparency, I think this will damage the concept of data independence. When an operating system is responsible, there will be a mismatch between requirements of the distributed DBMS and the functionality of the existing operating systems. Because, existing operating systems lack the functionality like distributed transaction support and efficient management of distributed data. Distributed DBMSs also require some modifications on operation systems traditional functions. In this context, it will be meaningless to embed too much database functionality inside the operating system kernel or to change traditional functions of operating system. In fact, it is clear that the convenience between the operating system and distributed DBMS is the most important one. The operating system should be flexible enough to support distributed DBMS functions and implement only the essential OS services and those DBMS functions that it can efficiently implement. In [Ozsu & Valduriez, 1991, pp. 71,72] this situation is explained as:

It is therefore quite important to realize that reasonable levels of transparency depend on different components within the data management environment. Network transparency can easily be handled by the distributed operating-system as part of its responsibilities for providing replication and fragmentation transparencies. The DBMS should be responsible for providing a high level of data independence together with replication and fragmentation transparencies. Finally the user interface can support a higher level of transparency not only in terms of a uniform access method to the data resources from within a language, but also in terms of structure constructs that permit the user to deal with objects in his or her environment rather than focusing on the details of database description.

3.4.2. Autonomy

There may be different forms of autonomy in an environment consisting of multiple points if these points have an idea of doing a cooperative work. These points

may be totally isolated from the environment they exist, therefore they are as free as possible in the shape of their work. They may be semi-autonomous, therefore while being aware of the environment they exist and having a cooperative work with other points in that environment, they may be free in some details of the work. In a tightly integrated environment, all the points have to do cooperation and they will never be free in some decisions although they have capability do to so.

In the context of distributed databases, autonomy indicates the degree to which individual DBMSs can operate independently. In tightly integrated systems, DBMSs can not operate independently, although they have the functionality. Thus an entire database is seen as a logically centralized database by the users. If a system is semi-autonomous, DBMSs can operate independently, but they can decide to participate in a federation to make some parts of their database accessible to other DBMSs. In a total isolation, there is no global control over the execution of individual DBMSs and therefore they are not aware of the existence of other DBMSs.

It will be difficult to execute user transactions involving multiple DBMSs if they are totally isolated because of the lack of global control. But there are also benefits of site autonomy. Some of the benefits of site autonomy are [Oracle 7]:

- Local data is controlled by the local administrator. Therefore, each administrator's region of responsibility is smaller and more manageable.
- Independent failures are less likely to disrupt other nodes of the distributed system. The global database is partially available as long as one database and the network are available. No single database failure need halt all global operations or be a performance bottleneck.
- Failure recovery is usually performed on an individual node basis.
- Nodes can upgrade software independently.

3.5. Architectural models for Distributed DBMSs

In Figure 3.5 taken from [Ozsu & Valduriez, 1991], different architectural models for distributed DBMSs are examined in three dimensions which are autonomy, distribution and heterogeneity. We know that autonomy means the distribution of control (tight integration, semi-autonomy and total isolation), distribution means the data distribution (either the data is distributed physically over multiple sites or it is kept at only one site), and heterogeneity means differences among individual DBMSs in terms of data models, query languages and transaction management protocols (two points are examined, in homogeneous systems each site has the same DBMS, in heterogeneous systems DBMSs are different).

In a distributed database, there must be three types of schema by which an integrated view of the individual databases is obtained. These are global schema, fragmentation schema and allocation schema. A global schema is a global view of all databases. A fragmentation schema is a one to many mapping between the global relations and fragments; and defines how relations are partitioned into fragments. The third one is an allocation schema, which is a mapping between fragments and sites; and determines the allocation of fragments into sites with or without replication. If this mapping is one to one, then the database is non-redundant, if the mapping is one to many, then the database is redundant.

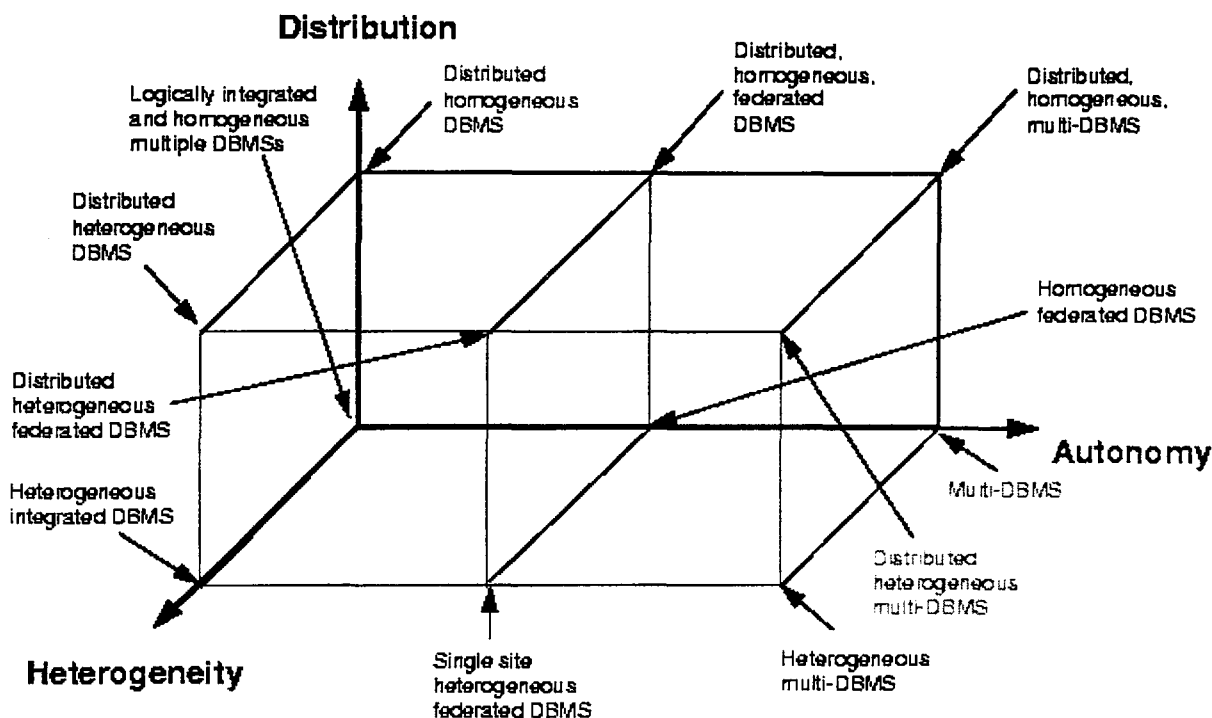


Figure 3.5 Architectural models for distributed DBMSs

In an autonomous environment, since each individual database is free, local access is done using a local DBMS and its schema. When remote access is necessary, then this is done through a global schema, which is an integrated view of the distributed database. If local databases are tightly integrated, this means that they are not permitted to use their local DBMSs individually, and without distinction between local or remote, all accesses are done through a global schema.

3.6. Query Processing

There are two main types of languages in which a query can be expressed, relational algebra and relational calculus. These are mathematical or pure relational languages and it is proved that both of them are equivalent in expressiveness. Relational algebra has a procedural nature and there is more than one way to write a query in relational algebra. Here, an important point is that, each different representation of the query results in a different execution cost. Therefore it requires a judgement to select one of the best way by considering the query execution cost.

However, relational calculus is more abstract and definitional which means that, the user only writes the requirements of the query, not the way it should be executed and let the responsibility of finding a procedure to execute the query, and return the result, to the query processor module of a DBMS. To give this simplicity to users in writing queries, non-procedural languages derived from relational calculus are developed. SQL (Structured Query Language) is a calculus-flavoured language that uses English language keywords. More information about the query languages can be found in [Ozkarahan, 1997].

After understanding the query written by a user in an abstract calculus-flavoured language, the query processor module has to decide the best way to execute it. Therefore it transforms the high level user query in relational calculus into a sequence of database operations on relational algebra. There are more than one equivalent and correct representations of a query in relational algebra, therefore query processor has to choose the best one that minimizes resource consumption. The query processing problem is much more difficult in distributed environments than in centralized ones, because not only choosing the best order of algebraic operations, but also a larger number of parameters affect the performance of distributed queries. In particular, the relations involved in a distributed query may be fragmented and/or replicated, thereby inducing communication overhead costs. The distributed query processor must select the best sites to process data, and possibly the way data should be transformed. This increases the solution space from which to choose the distributed execution strategy. Briefly, more than a centralized query processor, the role of a distributed query processor is transforming calculus query into an algebraic query described on relation fragments.

In [Ozsu & Valduriez, 1991], there are four layers of query processing. The first three layers are performed by a central site, and use global information; the fourth is done by the local sites:

1. **Query Decomposition:** This layer transforms the distributed calculus query into an algebraic query on global relations.

2. **Data Localization:** The input to the second layer is an algebraic query on global relations. In this layer, distributed algebraic queries expressed on global relations are mapped into queries on physical fragments of relations by translating relations into fragments. This process is called localization because its main function is to localize the data involved in the query. This layer determines which fragments are involved in the query and transforms the global query into a fragment query.
3. **Global Query Optimization:** This layer finds the best ordering of operations in the fragment query by considering the cost function which is a weighted combination of I/O, CPU, and communication costs. The decision to minimize communication cost includes data movement between sites and where each part of the query will be executed. Especially in wide area networks, where the limited bandwidth makes communication much more costly than local processing, I/O and CPU cost can be neglected to simplify the cost function.
4. **Local Query Optimization:** The last layer is performed by all the sites having fragments involved in the query. Each subquery executing at one site, called a local query, is then optimized using the local schema of the site. Local optimization uses the algorithms of centralized systems.

Among the layers of query processing, choosing the best point in the solution space of all possible execution strategies; namely query optimization, is the most important one. In choosing the best execution strategy, exhaustive search method is the most common one. This method exhaustively predicts the cost of each strategy, and selects the strategy with minimum cost. Although it is very efficient, it causes high optimization cost when the search space is large. To reduce the cost of exhaustive search, some heuristics are used. The goal of the heuristics is to restrict the solution space so that only a few strategies are considered. One of the most common heuristics is minimizing the size of intermediate relations. This is done by performing unary operations first and ordering the binary operations by the increasing sizes of their intermediate relations. Another important heuristic is replacing join operations by combinations of semijoins. Since the semijoin operation

has the important property of reducing the size of the operand relation, it is preferred especially in distributed environments to minimize data communication.

According to the time when the query optimization is made, there are two main types of query optimization, static and dynamic. Static query optimization is done at query compilation time, before the query is executed. Since the sizes of the intermediate relations are not known until run time, they must be estimated using database statistics. Thus the optimality of the strategy is depending on the correctness of the statistics. Dynamic query optimization is done at query execution time. At any point of execution, the choice of the best operation can be based on accurate knowledge of the results of the operations executed previously, therefore there is no need for database statistics. Of course these two types have some advantages and some disadvantages. First, since static query optimization is done before execution; it can be advantageous if that query will be executed more than once, thus amortizing the optimization cost. In dynamic query optimization we have to do optimization for every execution of the query, which is very expensive. On the other hand, while static query optimization is based on database statistics, which can have errors causing sub-optimal solutions, dynamic query optimization is eliminating these errors by depending on accurate results.

Providing the advantages of static query optimization while avoiding the issues generated by inaccurate estimates, hybrid query optimization is derived. The approach is basically static, but dynamic query optimization may take place at run time when a high difference between predicted sizes and actual size of intermediate relations is detected.

In query optimization, to choose an efficient execution plan, we need accurate estimates of the costs of alternative plans. One of the most important factors that affect plan cost is selectivity, which is the number of tuples satisfying a given predicate. Therefore, in most cases, the accuracy of selectivity estimates directly affects the choice of best plan. In [Chen & Roussopoulos, 1994], a method called “Adaptive Selectivity Estimation Using Query Feedback” is introduced. The goal of

this method is to approximate f_A (A is an attribute of relation R , and f_A is the actual distribution of A) by an easily evaluated function f which is able to selfadjust from subsequent query feedbacks. Thus, given a sequence of queries q_1, q_2, \dots , we can view f as a sequence f_0, f_1, f_2, \dots where f_{i-1} is used to estimate the selectivity of q_i and after q_i is optimized and executed, f_{i-1} is further adjusted into f_i using feedback (which contains the actual selectivity of query q_i after the execution).

3.7. Transaction Management

We need databases to store our meaningful data, then the database should have some rules to obey while doing any operations on the data. Users may perform wrong operations (updates, insertions and deletions) or wrong sequences of right operations on the data and these will introduce inconsistencies in the database. A database is consistent if it obeys all the consistency (integrity) constraints over it. Transaction management deals with the problems of always keeping the database in a consistent state even when concurrent accesses and failures occur.

While concurrency control and reliability protocols try to keep the database always in a consistent state, another fundamental issue concerning integrity constraints is **constraint checking**, that is the process of ensuring that the integrity constraints are satisfied after a transaction is completed over the database. The cost, which is associated with the performance of the checking mechanisms, is the main quantitative measure, which has to be supervised carefully. Different criteria have been used to assess this performance such as the time to check the validity of the constraints against updates. Generally, an efficient constraint checking strategy tries to minimize the utilization of the computing resources involved during the checking activities. A common goal addressed by previous researchers in this field is to propose constraint simplification strategies, which manage to derive a better set of constraints than the initial set. A simplification strategy is said to be efficient if the evaluation of the generated simplified forms has effectively reduced the cost of integrity checking compared to the evaluation of the initial constraints. The cost of evaluating an integrity constraint includes the following main components: (i) the

amount of data accessed - this is related to the checking space of the integrity constraint; (ii) the amount of data transferred across the network; and (iii) the number of sites involved.

In [Ibrahim et al., 1998], it is showed how the SICSDD (an integrity constraint subsystem for a relational distributed database, that they have developed) techniques have effectively reduced the cost of constraint checking in a distributed environment. Their techniques are **constraint preprocessing**, **constraint distribution** and **integrity test generation**. They are summarized below:

Constraint preprocessing techniques: There are five steps that are applied which are performed by the following procedures:

- **constraint transformation procedure:** transforms the constraints specification at the relational level into a constraints specification at the fragment level. Initially, each occurrence of a relation R in a constraint is replaced by its n fragment relations, R_i .
- **simplification procedure:** simplifies a set of fragment constraints if the relations involved in the constraints are fragmented on a join/reference attribute using the same fragmentation algorithm. This reduces the number of joins/references required during the evaluation of the constraint set.
- **subsumption procedure:** excludes redundant fragment constraints from a set. Even though a redundant set of constraints is semantically correct, excluding redundancy can improve the enforcement time.
- **contradiction procedure:** removes the fragment constraints produced by the constraint transformation procedure, which contradict the fragmentation rules.
- **reformulation procedure:** removes redundant semantic constructs that may exist and reformulates the set of fragment constraints into alternative forms which can be in equivalent form.

Constraint distribution techniques: Reduce the number of constraints allocated to each site by allocating a fragment constraint to a site if and only if there is a

fragment relation at that site which is mentioned in the constraint, so that whenever an update occurs at a site, the validation of the fragment constraints at that site implies the global validity of the update. Consequently, these techniques intend to allocate each fragment constraint to a site or minimal number of sites and relieve the irrelevant sites from the computation of certain sets of fragment constraints.

Integrity test generation techniques: Generate integrity tests from the syntactic structure of the constraints and the update operations. The algorithms applied for deriving these tests use the substitution, subsumption and absorption rules.

A detailed information can be found at [Ibrahim et al., 1998].

3.7.1. Introduction to Transactions

A transaction is a basic unit of consistent and reliable computation. Similar to the query, a transaction performs an action on the database and generates a new version of it, therefore causing a state transition from one state of the database to the other. An important point here is, while we can not ensure that an execution of the query will result in a consistent new state of the database; in a transaction this is ensured. Therefore, a transaction may be thought of as a program with embedded database access queries.

If the transaction can complete its task successfully, we say that the transaction commits, otherwise it aborts. When a transaction is aborted, its execution is stopped and all of its already executed actions are undone by returning the database to the state before their execution. This situation is called as rollback. A transaction may abort itself if a condition exist that prevent it from completing its task successfully; or the DBMS may abort a transaction because of the deadlocks or other conditions. When the commit command is issued, the DBMS understands that the effects of the transaction will be reflected in the database, and once a transaction is committed, the results are permanently stored in the database and cannot be undone.

There are four properties of transactions; **atomicity, consistency, isolation and durability**. These properties are main factors for a transaction to ensure database consistency and reliability. Atomicity means that either all actions of a transaction are completed or none of them are. In other words, it refers to the fact that a transaction itself, in its entirety is a unit of operation. If the execution of the transaction is interrupted by any sort of failure, DBMS would be responsible for either completing the remaining actions after recovery or terminating it by undoing all the actions that have been executed by the help of reliability protocols. Consistency means the correctness of a transaction meaning the transition from one consistent database state to another. Ensuring transaction consistency is the responsibility of concurrency control mechanisms. Isolation is a property of transactions, requiring the condition that an executing transaction cannot reveal its results to other concurrent transactions before its commitment. If two concurrent transactions can reach the same data item that is just updated by one of them, there is a big possibility that the second one will get incorrect values, because their operations won't be isolated and interfere with each other. By this property, incomplete results are not permitted to be seen by other transactions preventing the lost updates. Besides the lost updates problem, if one transaction's incompleting results are seen by others, and the transaction aborts, then all the other transactions which have used incomplete results have to abort as well. This situation is called as cascading aborts. Providing isolation property to transactions is also the responsibility of concurrency control mechanisms. The last property of transactions is durability meaning that the results of a committed transaction are permanent and cannot be erased from the database even there is a site failure. In obtaining durability, again reliability protocols have an important role.

3.7.2. Concurrency Control Protocols

Concurrency control deals with the isolation and consistency properties of transactions. When a concurrency control will be valid for a distributed database, we have to care about partitions and replicas of data items also, therefore the problems of concurrency control in a distributed DBMS are more severe than in a centralized

DBMS. In fact, if we execute each transaction alone, one after another, there will be no inconsistency, but we should also think of the throughput of the system, therefore we should consider concurrent accesses and find methodologies to solve the consistency problem. There are two major classes of concurrency control algorithms: locking-based and timestamp-based.

Locking-based Concurrency Control Algorithms

In the locking-based algorithms, the synchronization of transactions are achieved by employing physical or logical locks on the portion of the database, which is being accessed by a transaction. Therefore, when any portion of the database is locked by a transaction, other concurrent transactions have to wait to access that portion until the transaction holding the lock finishes its execution. The termination of a transaction results in the release of its locks and initiation of another transaction that might be waiting for access to the same data item.

When we aim increased throughput, it seems advantageous for a transaction to release its locks on any data item, as soon as it finishes operation on it, although it is not terminated yet. But this will violate the isolation and atomicity properties of transactions, since transactions will interfere with each other. Therefore, two-phase-locking algorithm seems to be a solution.

Two-phase-locking (2PL) algorithm

This algorithm states that, a transaction should not release a lock until it is certain that it will not request another lock. To ensure this rule, 2PL algorithm execute transactions in two phases which are growing phase where a transaction obtains all the locks it needs, and shrinking phase, in which all the obtained locks are released. The point between these two phases, in other words, the point in which a transaction obtains all of the locks but has not yet released any of them is called the lock point.

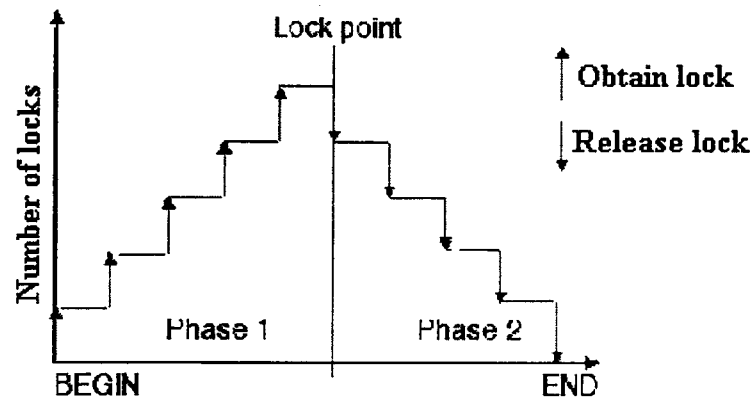


Figure 3.6 2PL Lock Graph

As shown in Figure 3.6, in this algorithm, a transaction begins to release its locks in second phase, after obtaining all the locks, finishing operation on them and it is sure that no other locks won't be needed. This means that in the second phase of a transaction, other transactions can obtain its locks, therefore increasing the degree of concurrency. But here, another difficulty, which is the determination of lock point arises. Besides this difficulty, if the transaction aborts after it releases a lock, other transactions that have accessed the unlocked data item have to abort as well, causing cascading aborts. The solution could be strict two-phase locking.

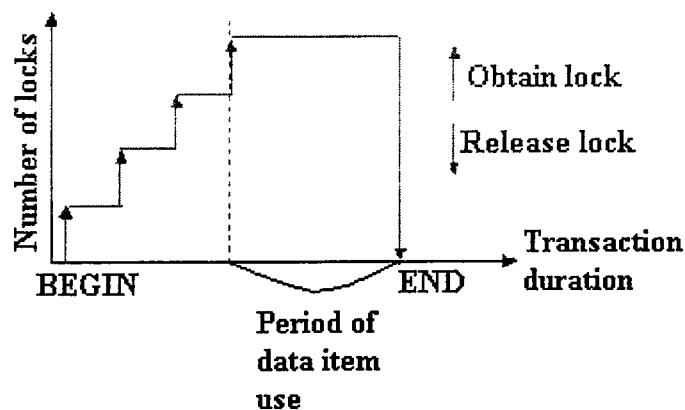


Figure 3.7 Strict 2PL Lock Graph

As can be seen from Figure 3.7, in this algorithm, a transaction releases all the locks together when the transaction terminates (commits or aborts). This algorithm can be applied in distributed database environment with some modifications. The simplest way of doing this is, choosing one site as a lock manager having the responsibility of all the locks in an entire distributed database.

Here, when a distributed transaction is initiated at one site, the transaction manager (TM) at that site becomes coordinating TM, and sends a lock request to a central lock manager. If lock manager grants the lock, it sends “lock granted” message to coordinating TM. Having granted the lock, coordinating TM now free to distribute the transaction to participating sites. When participating sites finish their operation, they send “end of operation” message to coordinating TM. After getting this message from all the participating sites, coordinating TM informs the lock manager (LM) that the transaction is terminated, by sending “release locks” message. The communication structure can be seen in Figure 3.8.

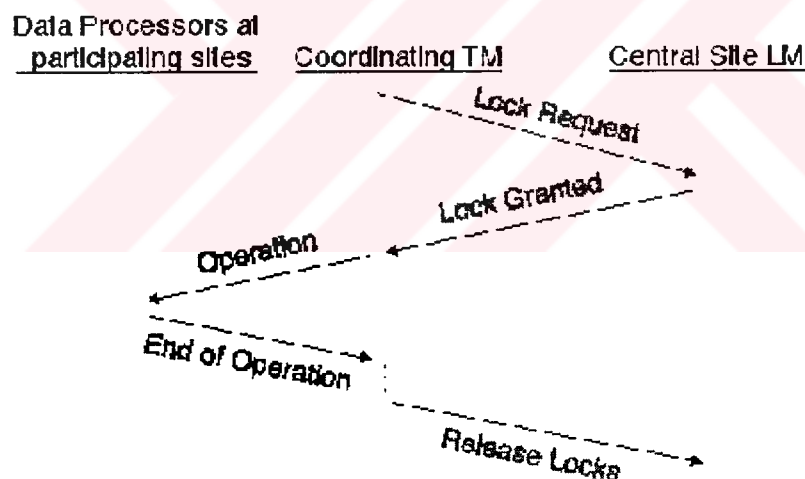


Figure 3.8 Communication Structure of Centralized 2PL

Although this is a usable algorithm in a distributed database environment, it shares the main disadvantage of central algorithms, being single point of failure. The elimination of this disadvantage can be made by using distributed 2PL algorithm (Figure 3.9). In this derivation of algorithm, all sites are available to be a lock

manager, and each of them are responsible for lock requests for data at their own sites. A transaction may read any of the replicated copies of item x , by obtaining a read lock on one of the copies of x , writing on x requires obtaining write locks for all copies of x (ROWA rule).

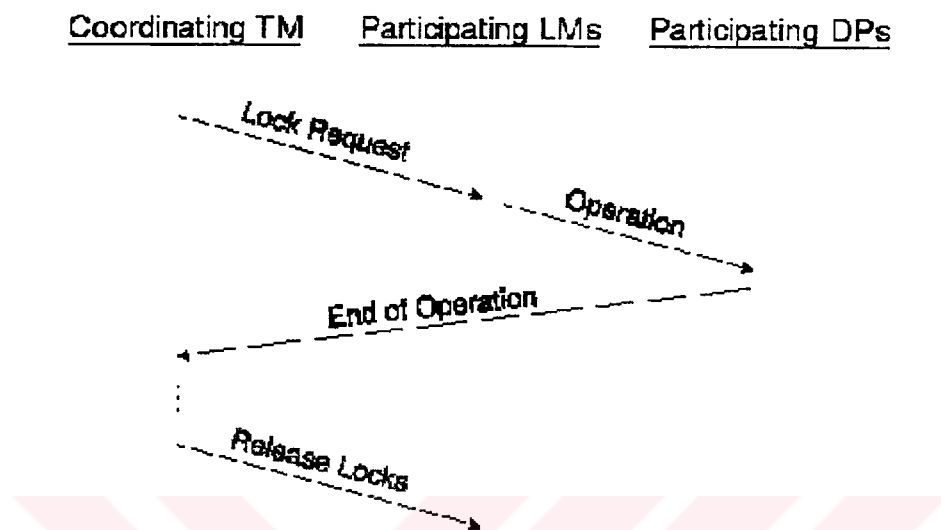


Figure 3.9 Communication Structure of Distributed 2PL

Although it is simple to implement locking-based algorithms, these may result in deadlocks. A deadlock can be defined as a situation, in which transaction T_1 waiting for item A_1 held by T_2 , which is waiting for A_2 held by T_3 , and so on, while T_k is waiting for A_k held by T_1 ; which implies a cycle in the waiting for relationship [Ullman, 1982]. Therefore, timestamp-based concurrency algorithms are generally preferred.

Timestamp-based Concurrency Control Algorithms

Basically, all transactions are given a unique timestamp and this timestamp is attached to all operations of the transaction. Therefore transactions are ordered according to these timestamps. Timestamps are selected from a totally ordered domain. In a centralized environment, unique timestamps can be obtained simply by assigning a counter value. But in a distributed environment, a unique timestamp can be date/time/site combination, or more formally $\langle \text{local counter value, site identifier} \rangle$.

According to these unique timestamps of transactions, which require access to the same piece of data, the database management system uses one of a number of protocols to schedule them. In [Ozsu & Valduriez, 1991, p.299], the timestamp ordering (TO) rule is given as:

“Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k , O_{ij} is executed before O_{kl} if and only if $ts(T_i) < ts(T_k)$. In this case T_i is said to be the older transaction and T_k is said to be the younger one.”

In this definition $ts(T_i)$ means the timestamp of the transaction T_i . The basic TO algorithm is a simple implementation of the TO rule. In [Ozsu & Valduriez_notes. 1999], the basic TO algorithm is described as follows: Each data item is assigned a write timestamp (wts) and a read timestamp (rts). $rts(x)$ is the largest timestamp of any read on x and $wts(x)$ is the largest timestamp of any write on x . $R_i(x)$ is the read operation of transaction T_i , and $W_i(x)$ is the write operation of transaction T_i . According to these definitions:

for an operation $R_i(x)$,
if $ts(T_i) < wts(x)$ **then** reject $R_i(x)$
else accept $R_i(x)$ and $rts(x) \leftarrow ts(T_i)$

for an operation $W_i(x)$,
if $ts(T_i) < rts(x)$ and $ts(T_i) < wts(x)$ **then** reject $W_i(x)$
else accept $W_i(x)$ and $wts(x) \leftarrow ts(T_i)$

A transaction's operation that is rejected is restarted by the transaction manager with a new timestamp, therefore has a chance to execute it in the next try and by this way basic TO algorithm never causes deadlocks.

3.7.3. Reliability Protocols

The aim of reliability protocols is maintaining the atomicity and durability of distributed transactions that execute over a number of databases. Commit, termination and recovery protocols are the reliability protocols that exist in a distributed database environment.

We know that, in the case of distributed transactions, the global transaction contains several local transactions. The global transaction is only successful if all of the local transactions are successful. Commit protocols ensure that a global transaction is either successfully completed at each site or aborted. One node among the participating transaction nodes should play the role of a coordinator node. The coordinator node is responsible for taking the final commit or abort decision.

During the execution of a distributed transaction, some sites might fail. Although there is a possibility of failures in participating sites, commit protocol synchronizes the effects of the transaction among multiple sites; therefore the effect of the distributed transaction on the distributed database should be all-or-nothing. In the case of failures, besides commit protocol, there are two other protocols that are necessary, termination and recovery protocols. These protocols are interested in two faces of a failure problem. While termination protocols address how operating sites will be affected from failure, recovery protocols address what the failed site should do when it is recovered and restarted.

Two-Phase Commit Protocol

In [Ozsu & Valduriez, 1991, p349], a brief description of the 2PC protocol that does not consider failures is as follows:

Initially, the coordinator writes a begin-commit record in its log, sends a "prepare" message to all participant sites, and enters the WAIT state. When a participant receives a "prepare" message, it checks if it could commit the

transaction. If so, the participant writes ready record in the log, sends a “vote-commit” message to the coordinator, and enters READY state; otherwise, the participant writes an abort record and sends a “vote-abort” message to the coordinator. After the coordinator has received a reply from every participant, it decides whether to commit or to abort the transaction. If even one participant has registered a negative vote, the coordinator has to abort the transaction globally. So it writes an abort record, sends a “global-abort” message to all participant sites, and enters the ABORT state; otherwise, it writes a commit record, sends a “global-commit” message to all participants, and enters the COMMIT state. The participants either commit or abort the transaction according to the coordinator’s instructions and send back an acknowledgement, at which point the coordinator terminates the transaction by writing an end-of-transaction record in the log.

This is called centralized 2PC protocol since the communication is between the coordinator and participants only. In this process, participants do not communicate among themselves. Coordinator site is the site, where the transaction is initiated. In Figure 3.10, centralized 2PC communication structure and in Figure 3.11 state transitions for both coordinator and participating sites can be seen.

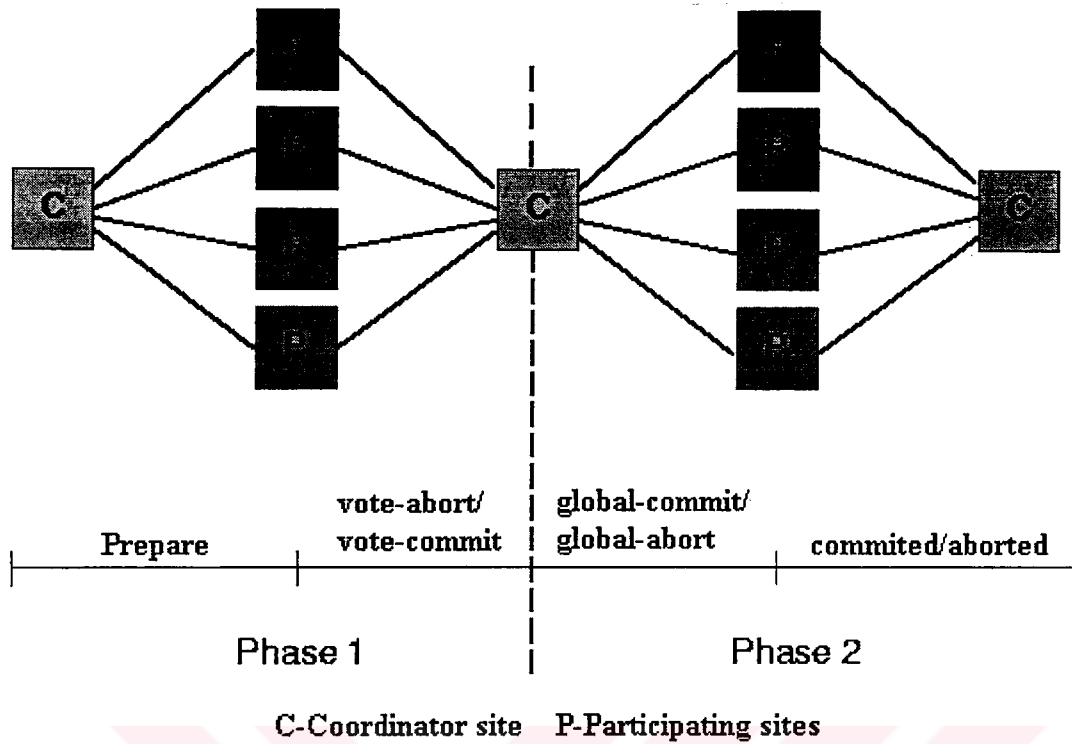


Figure 3.10 Centralized 2PC Communication Structure

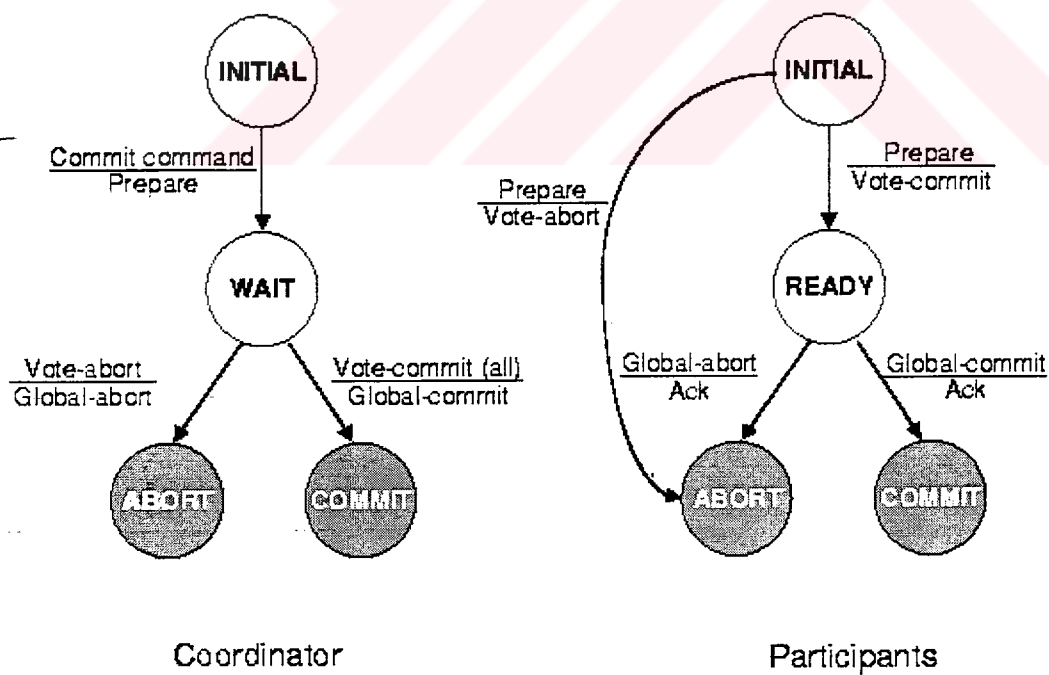


Figure 3.11 State Transitions in 2PC Protocol

The major disadvantage of the two-phase commit protocol is the fact it is a blocking protocol. Because a site will be blocked while it is waiting for a message from another site. This means that, other processes competing for resource locks, held by the blocked process will have to wait for the locks to be released. This can cause dead-locking.

Another disadvantage of the two-phase commit protocol is the fact a transaction can hang if a participant site fails while the coordinator is in the ABORT or COMMIT state -the coordinator is unsure if the transaction has been committed or aborted. Therefore, it has to wait. If a site is down permanently, this will cause the coordinator to wait forever and the transaction never gets resolved. On the other hand, a transaction can also hang if the coordinator goes down and the participant is in the READY state. The participant can't unilaterally make a decision to "commit" or "abort", therefore it remains blocked until the coordinator comes back on-line and replies "ABORT" or "COMMIT". Else, it will wait forever and the transaction never gets resolved.

Three-Phase Commit Protocol

Because of the disadvantage of blocking, Three-Phase Commit (3PC) protocol is designed. The 3PC algorithm is an atomic non-blocking algorithm. A non-blocking algorithm enables a decision (commit or abort transaction) to be reached at every participant or at the coordinator despite the permanent failure of other sites (unlike 2PC). Non-blocking protocols are also desirable because they limit the time intervals during which transactions may be holding valuable resources. The 3PC contains an additional state, which is between the **ready to commit** state and the **committed** state. This state is called **prepared to commit** state. The addition of this intermediate state eliminates blocking since

- There is no state that is "adjacent" to both a commit and an abort state
- No noncommittable state that is "adjacent" to a commit state.

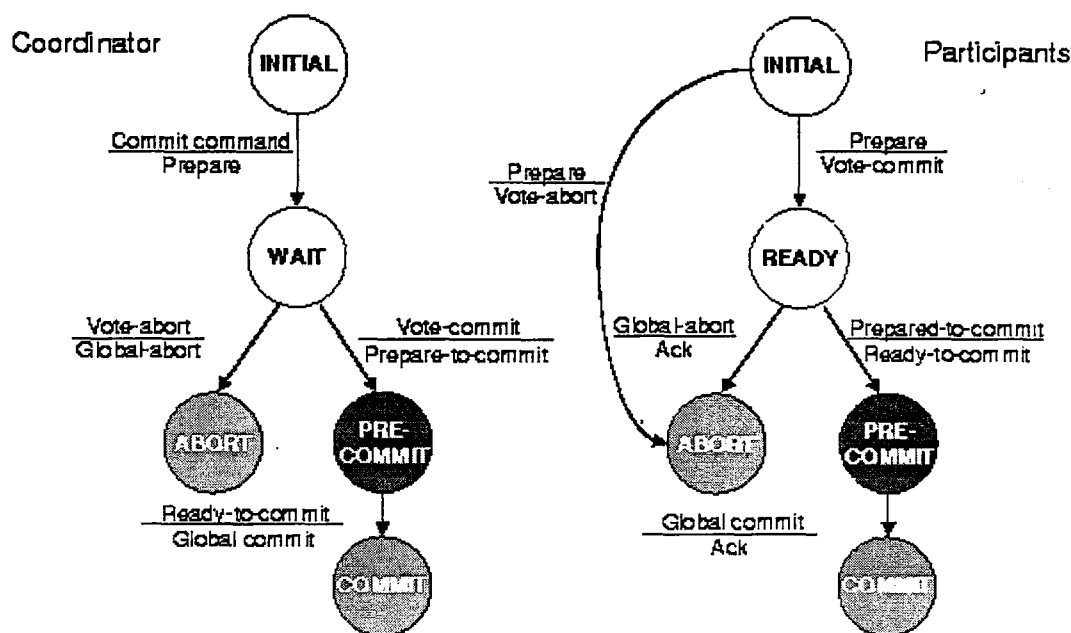


Figure 3.12 State Transitions in 3PC Protocol

More detailed information about termination and recovery protocols used in 2PC and 3PC protocols can be found in [Ozsu & Valduriez, 1991]. Although 3PC protocol eliminates the blocking problem, it involves an extra round of message transmission, which further degrades the performance of DDBSs. In [Reddy & Kitsuregawa, 1998], a backup commit (BC) protocol is proposed by including backup phase to 2PC protocol. In this protocol, one backup site is attached to each coordinator site. After receiving responses from all participants in the first phase, the coordinator communicates its decision only to its backup site in the backup phase. Afterwards, it sends final decision to participants. When blocking occurs due to the failure of the coordinator site, the participant sites consult coordinator's backup site and follow termination protocols. In this way, BC protocol achieves non-blocking property in most of the coordinator site failures. However, in the worst case, the blocking can occur in BC protocol when both the coordinator and its backup site fail simultaneously. If such a rare case occurs, the participants wait until the recovery of either the coordinator site or the backup site. BC protocol suits best for DDBS environments in which sites fail frequently and messages take longer delivery time.

Through simulation experiments it has been shown that BC protocol exhibits superior throughput and response time performance over 3PC protocol and performs closely with 2PC protocol.

In [Zhang et al., 1998], a novel approach to atomic commitment of transactions in distributed database systems is proposed. This approach charges two active nodes, which are called Master Active Node (MAN) and Secondary Active Node (SAN), by maintaining passive "blackboards" which record voting status of participant subtransactions. These active nodes are participating in the process of atomic commitment of distributed transactions.

We know that 2-phase commit and its derivatives have the synchronization rule that, the global transaction is successful only when all of the local transactions are successful, therefore these protocols ensure that a global transaction is either successfully completed at each site or aborted. Because of this rule, these protocols represent synchronous replication, which couples together all updates to all locations participating in an update. This becomes difficult as the number of participating nodes increases, thus 2-phase commit and its derivatives for updating will be probably impractical. Any failure in the network or any of the local participating databases causes the entire transaction to fail, which is very intolerant to failure. Because of these penalties, different replication approaches are introduced. In these approaches, the timing between the changes at the different nodes is managed through mail or store and forward approaches rather than through locked multi-site transactions. Once the application updates its local data, it is decoupled from the replication engine, which has the responsibility for propagating the copies of the changed data to other locations. A transaction managed through a replication approach is considered successful if it is committed at one site. Although these approaches are more fault tolerant and therefore more appropriate for many applications, they can not be used where absolute data synchronization is required for the application. Examples of such applications would be financial trading and banking funds transfer. If the application can deal with some inconsistency among the different data nodes for short periods of time, then replication should be

considered as an alternative. Different replication approaches used by some products of current technology can be found in Chapter 5.1.



CHAPTER FOUR

DIFFERENT CLIENT-SERVER DATABASE SYSTEMS

The general Client-Server model (refer to chapter 2) for network applications can be easily extended to database systems. The central concept in Client-Server Database Systems (CS-DBMSs) is that a dedicated machine runs a DBMS and maintains a main centralized database (DBMS-Server). The users of the system access the database through either their workstations or PCs via the network. They usually run their programs locally on their own machines and direct all database inquiries and/or updates to the DBMS-Server. In this way, they become the server's clients. This configuration is termed Standard Client-Server DBMS (SCS) [Delis & Roussopoulos, 1994]. SCS is the central system, since the database is located at only one machine and all clients from other machines access it via the network. Different implementations of SCS from [Delis & Roussopoulos, 1992] are given in section 4.1.

In this model, client and server have different roles. Client's roles are, managing the user interface, accepting data from user, processing application logic, transmitting database request to server, receiving results from server and formatting results. On the other hand, server's roles are, accepting database request from clients, processing database requests, formatting result and transmitting to client, providing concurrent access control, performing recovery and optimizing query/update processing.

The relationship between the client and the server is different in distributed database systems, a system might act as a server in one transaction and as a client in

another. In a distributed client/server relationship, both the client and the server (hardware and software alike) have data-repository and database-processing duties. The implementation of client-server model in distributed databases is given in section 4.2.

4.1.Client-Server Architecture for Centralized Databases

The SCS architecture off-loads CPU processing from the servers to the clients. The application programs along with other interface utilities, such as the DBMS presentation manager, are run on the clients without affecting the server. The bulk of the database processing and I/O remains a server task. Figure 4.1 presents a cluster of clients with a single server.

To give better response time and increase the system throughput, Client-Server with Multiple Disks (CS-MD) architecture is developed. The CS-MD architecture incorporates a large number of disks on the servers and intelligent controllers, which utilize multiple heads for reading in parallel from replicated data (Figure 4.2). We assume that each disk has a copy of the database -full replication-, since it provides a practical approach to the data allocation problem. The job management uses a locking mechanism similar to that of the SCS concurrent servers with the only exception that it uses the read-one/write-all protocol, that is, an update commits only when all disks have finished the update, and a read is done from a single disk, the one which is idle or has the lightest load. This configuration favours reads at the expense of updates but avoids the overhead of partial replication and skewed access patterns.

The major advantage of the CS-MD architecture over SCS is that it distributes read operations over a number of disks yielding better response times and ultimately increased system throughput. However, write operations may create additional conflicts, more blocking, and increased overhead.

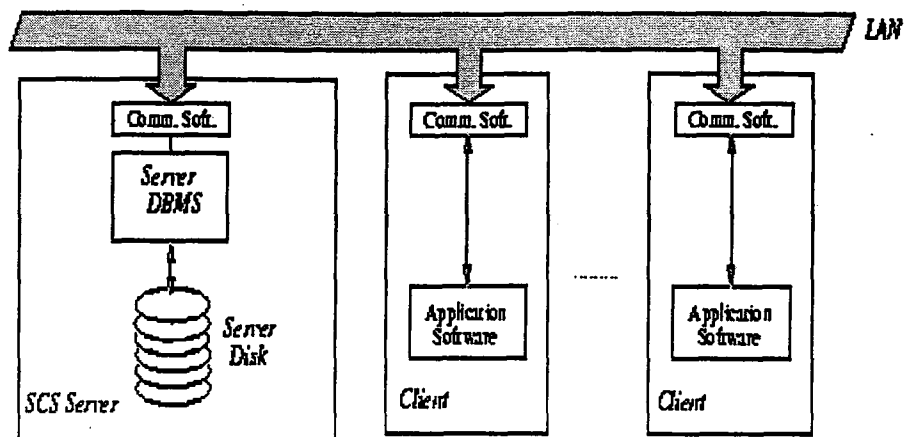


Figure 4.1 SCS Architecture

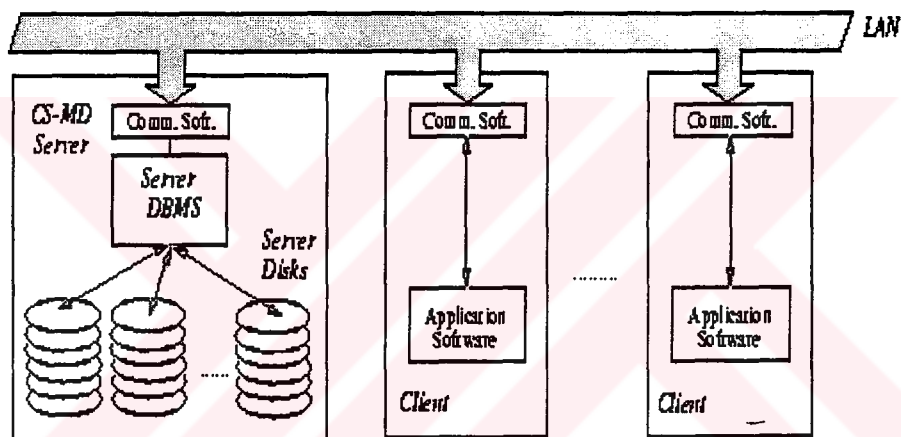


Figure 4.2 CS-MD Architecture

The above two architectures centralize the database operations on the servers and distribute to the clients only application/interface processing. The Enhanced Client-Server (E-CS) architecture goes further and distributes to the clients a portion of database operations. Therefore The Enhanced Client-Server DBMS off-loads disk accesses from the server by having the clients run a limited DBMS, in terms of concurrency, and by caching results of server queries to the client disk managers. Therefore, this model is so important, since it prepares the basics for distributed client/server database architecture.

To achieve this, it utilizes the local disks available on the client workstations for caching query results once retrieved from the servers and delivered to the clients. The additional merit is that the clients' disks are accessed asynchronously contributing to greater I/O parallelism whenever this is possible. This architecture requires disk cache management functionality on the clients for dynamic data migration and incremental maintenance or replacement of cached data. This functionality is very similar to that of a DBMS except that a) it needs no transaction recovery and security managers (each client user runs in his/her own locally cached environment), and b) it is capable of handling cached query results, which are bound to server(s) relations. The E-CS architecture is depicted in Figure 4.3.

Initially, the clients can start with either an empty local cache or with some data of their interest. Queries involving server relations are transmitted to and processed by the server(s). Their results -in the format of tuples- are shipped to the appropriate clients for displaying and/or other processing. Clients can then cache these results in local relations on their disk for later use. At that time, a binding between a client and the server is created. The binding in the format of query conditions and a timestamp is stored in the catalog of the client. Bound cached query results are the product of selections/projections/joins from the server relations. Dynamic caching permits the clients to define their "operational database space" according to their needs and constitutes a form of replication. It is reasonable to assume that most clients would be interested in a subset of the database space, therefore, the degree of replication in N clients would be less (or a lot less) than N copies (full replication) of the whole database space.

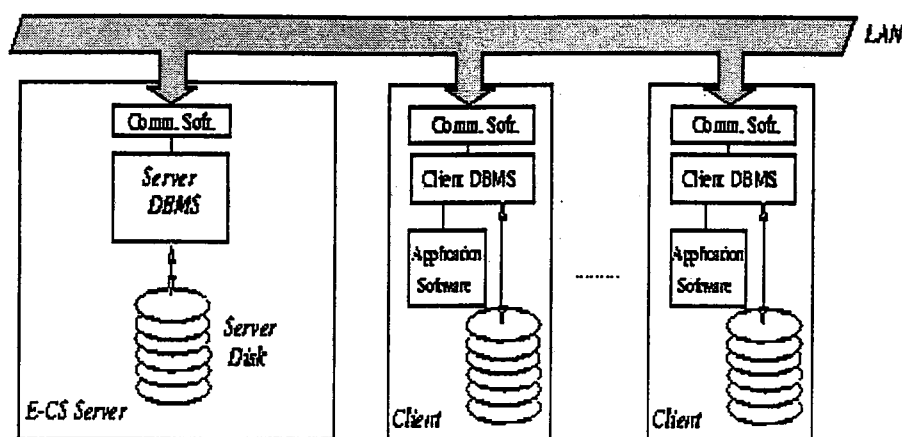


Figure 4.3 E-CS Architecture

4.2. Client-Server Architecture for Distributed Databases

Unlike centralized client/server relationships, in which the database resides entirely on the server, in distributed client/server relationships, however, both the client and the server maintain a portion of the database (data and processing) and alternate between acting as client and as server. In a distributed database, one server may need to access a database on another server. In this case, the server requesting the information becomes a client.

We can explain this situation with an example from [Oracle 7]. In Figure 4.4, the computer that manages the HQ database is acting as a database server when a statement is issued against its own data (for example, the second statement in each transaction issues a query against the local DEPT table), and is acting as a client when it issues a statement against remote data (for example, the first statement in each transaction is issued against the remote table EMP in the SALES database).

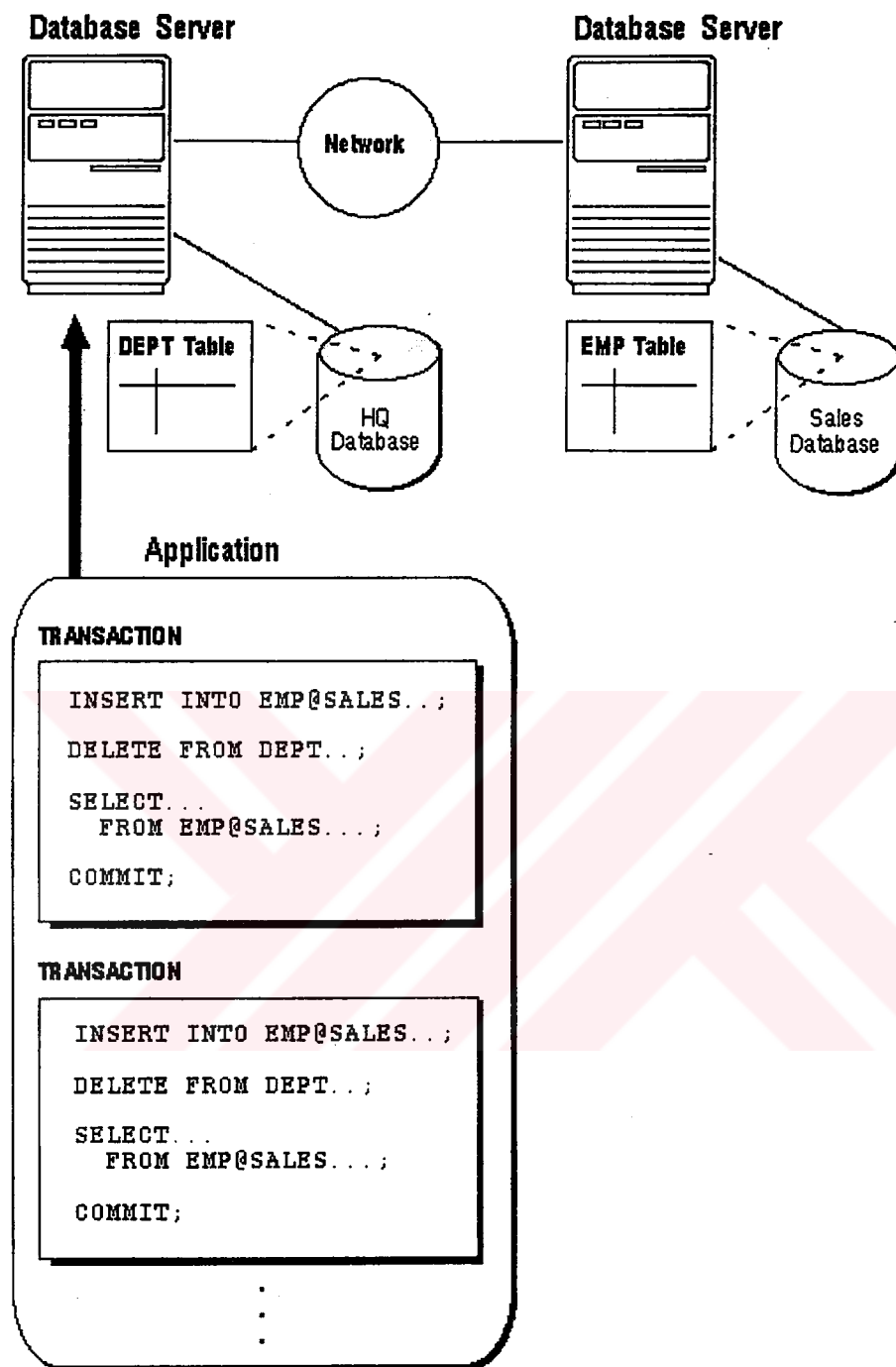


Figure 4.4 An Example of a Distributed DBMS Architecture

CHAPTER FIVE

THE CURRENT STATE AND FUTURE EXPECTATIONS

Up to this point, the necessary information to answer the question “What were the initial promises and goals of the distributed database technology?” was given. In this chapter the answers to the below questions will be examined :

- In section one, “How do the current products measure up to the promises of distributed database technology?”
- In section two, “Are the goals of distributed database technology achievable, and what are the unsolved problems?”
- In section three, “What are future expectations?”

5.1. The state of the current products

A first-generation of distributed solutions require conversion of all the databases, regardless of the platform, to run under the same DBMS program. Since the cost of converting to a homogeneous, single-vendor solution is too expensive; these offerings represent a critical first step toward achieving true distributed-database systems.

Second-generation products solve many of the short-term issues with heterogeneous distributed-database implementations including cross-server and cross-platform data access and security, and integration of distributed and nondistributed systems. New and upcoming releases from the DBMS vendors

address many of these issues. Additionally, data-warehousing tools, such as Dynamic Information Systems' Omnidex; middleware, including drivers, routers, and gateways from companies ranging from Borland to IBM; integrated data-access tools, such as the Personal Series from Uniface and Information Builders' EDA/SQL; and integrated programming tools, such as Cognos's PowerHouse, Open Environment's OEC Toolkits, and Inference's ART*Enterprise all provide unique solutions for data access, security, and integration [Richter 1994].

These second-generation products provide more flexibility in terms of which systems and DBMS products can be part of the distributed network. However, many of the technical issues surrounding true transparent distributed databases remain unsolved in today's commercial products. Nevertheless, the business demand for client/server and enterprise-wide computing will almost certainly continue to drive distributed-database technology to more refined and sophisticated heights.

In [Richter 1994], how different products solve data location and transparency problems and their approaches to commit protocols had been examined briefly:

How different products solve the data location and transparency problem?

- Cincom's Supra Server uses DRDM (Distributed Relational Data Manager) technology to solve the problem of data location. The DRDM includes a distributed metadata catalog as well as rules for handling changes across the system. The metadata catalog includes the key information catalog plus information about the physical site of the data. All subsequent database operations (distributed or not) run against the DRDM, thus ensuring accurate data.
- Sybase uses its System 10 OmniSQL Gateway interface to provide location transparency. The OmniSQL Gateway centrally stores a single global catalog that maps data and other database entities to their distributed locations. All distributed transactions run through the gateway.

- Data-warehouse solutions such as Omnidex, maintain and use a central repository of indexed data structures and keywords for rapid data location and extraction. The drawback to warehousing is that it is read-only and is typically updated only periodically.
- Oracle stores a central catalog of directories (i.e., data locations) but local catalogs of data (i.e., data dictionaries).

Approaches that take place of 2PC

Because of some shortcomings of 2PC protocol discussed before, vendors have moved beyond the traditional two-phase commit.

- Supra Server uses a form of retroactive polling to compensate for a failed commit. Once a transaction begins (i.e., all sites confirm availability for a transaction), all involved sites become transaction partners. The coordinating site posts a list of partners to each of the participating sites. If a site becomes unavailable for the second phase, Supra Server may, at the discretion of the remaining partners roll back the entire transaction, or it may ignore the inactive site and post the changes to the active sites. In the latter case, when the off-line site becomes available, the DRDM alerts it to changes, and it falls to the newly restored site to poll the other partners for the updated information.
- Sybase 10 and Oracle 7 “batch” the changes for the inactive site and post them when the inactive site comes back on-line, using a technique called store and forward. For example, Sybase SQL Server, using data replication implemented through the Sybase 10 Replication Server, selects a primary site as the data keeper. All changes to the data run against the primary site. When the primary data changes, the Replication Server automatically and transparently updates the replicated data across the distributed system. If a particular site is unavailable, the Replication Server queues the transactions and posts them as soon as possible.

Now, different approaches of different vendors will be examined in more detail.

5.1.1. Oracle

The information below is taken from [Oracle7].

The Oracle Client/Server Architecture

In the Oracle client/server architecture, the database application and the database are separated into two parts: a front-end or client portion, and a back-end or server portion. The client executes the database application that accesses database information and interacts with a user through the keyboard, screen, and pointing device such as a mouse. The server executes the Oracle software and handles the functions required for concurrent, shared data access to an Oracle database.

Although the client application and Oracle can be executed on the same computer, it may be more efficient and effective when the client portion(s) and server portion are executed by different computers connected via a network. The following are examples of distributed processing in Oracle database systems:

- The client and server are located on different computers; these computers are connected via a network (see Figure 5.1, Part A).
- A single computer has more than one processor, and different processors separate the execution of the client application from Oracle (see Figure 5.1, Part B).

Benefits of the Oracle client/server architecture in a distributed processing environment include the following:

- Client applications are not responsible for performing any data processing. Client applications can concentrate on requesting input from users, requesting desired data from the server, and then analyzing and presenting this data using the display capabilities of the client workstation or the terminal (for example, using graphics or spreadsheets).

- Client applications can be designed with no dependence on the physical location of the data. If the data is moved or distributed to other database servers, the application continues to function with little or no modification.

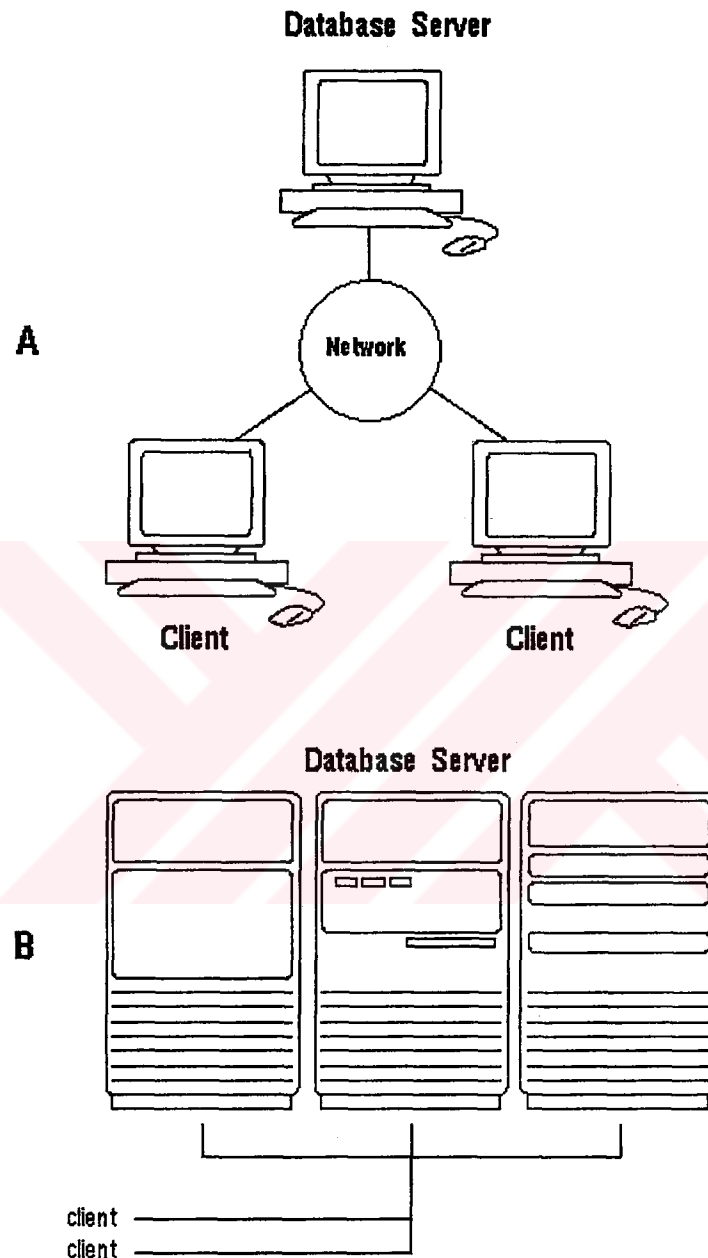


Figure 5.1 The Client/Server Architecture and Distributed Processing

- Oracle exploits the multitasking and shared-memory facilities of its underlying operating system. As a result, it delivers a high possible degree of concurrency, data integrity, and performance to its client applications.
- Client workstations or terminals can be optimized for the presentation of data (for example, by providing graphics and mouse support) and the server can be optimized for the processing and storage of data (for example, by having large amounts of memory and disk space).
- If necessary, Oracle can be scaled. As the system grows, multiple servers can be added to distribute the database processing load throughout the network (horizontally scaled). Alternatively, Oracle can be replaced on a less powerful computer, such as a microcomputer, with Oracle running on a minicomputer or mainframe, to take advantage of a larger system's performance (vertically scaled). In either case, all data and applications are maintained with little or no modification, since Oracle is portable between systems.
- In networked environments, shared data is stored on the servers, rather than on all computers in the system. This makes it easier and more efficient to manage concurrent access.
- In networked environments, inexpensive, low-end client workstations can be used to access the remote data of the server effectively.
- In networked environments, client applications submit database requests to the server using SQL statements. Once received, the SQL statement is processed by the server, and the results are returned to the client application. Network traffic is kept to a minimum because only the requests and the results are shipped over the network.

Oracle Distributed Database Technology

Oracle is offering solutions for both synchronus and asynchronous environments. In [Oracle 7], the offerings of synchronous capability, and asynchronous capability is given. Briefly, they will be mentioned below :

Oracle7 Synchronous Distributed Capability Overview

- **Distributed Queries** Data on multiple databases can be queried using the full functionality of the SQL standard SELECT statement. This includes selection, join, aggregation, and sorting operations optimized for maximum performance in a distributed environment.
- **Distributed Transactions** Data on multiple databases can be modified using the full functionality of SQL standard UPDATE, DELETE, and INSERT statements operating as transactions to ensure that either all modifications on all databases complete successfully or are all rolled back should failures occur.
- **Remote Procedure Calls** Oracle7 servers can execute remote PL/SQL procedures on other servers. The remote procedure execution operates within the same transaction again ensuring that either all modifications complete or are all rolled back.
- **Synchronous Replication** PL/SQL triggers can apply all modifications to tables in one database directly to replicate copies of those tables in other databases. All modifications are applied within the same transaction to ensure exact, point-in- time consistency of all copies.
- **Location Transparency** Applications can access remote data and execute remote procedures as easily as they do locally. No special coding is required to specify data or procedure location. Applications merely specify the data and procedures they need to access using logical names. The mapping from logical names to exact physical locations is done transparently by the database. Data and procedures can be moved from one database to another without requiring that application code be modified.

- **Commit Transparency** Applications can execute distributed transactions across multiple databases as easily as local transactions. No special coding is required. The SQL standard COMMIT statement commits both local and distributed transactions transparently.
- **Robust Protection Against Failures** Systems will fail and networks will fail. Oracle7 protects the integrity of distributed transactions automatically using a robust underlying two phase commit protocol. Complex two phase commit logic does not need to be coded into applications.
- **Status Information** The status of distributed transactions can be easily obtained and monitored through standard data dictionary tables within the database using Oracle Server Manager and other tools.
- **Single Server** No additional, special servers need to be installed and maintained.
- **Direct Server-Server Communications** Distributed operations are performed automatically through direct server to server connections. Applications do not need additional connections to external servers to perform operations such as distributed queries. Distributed data access does not need to pass through extra servers impacting performance.
- **Compatibility** Operations such as procedure calls operate under the same transactional protections whether they operate locally or remotely.

Oracle7 Asynchronous Distributed Capability Overview

Oracle's first asynchronous distributed capability was read-only snapshots, which provides a basic asynchronous replication capability. One-site, the snapshot master, can be updated. All other replicates or snapshots are read-only. Incremental row changes are propagated using a fast refresh mechanism. The snapshot refresh group feature ensures transactional consistency to maintain referential integrity between multiple snapshots. Read-only snapshots are also very easy to create and administer. Based on the experience gained from this first offering, Oracle introduced its next generation asynchronous distributed technology called "Symmetric Replication". The capabilities of symmetric replication are:

- **Basic and Advanced Replication Support** Asynchronous replication usage models are methodologies for conflict avoidance or conflict detection and resolution. Applications employing asynchronous replication use one or more of these methodologies to ensure data convergence. Symmetric Replication is designed to support replication models employing both conflict avoidance and conflict detection and resolution usage methodologies. Unlike other replication products, the underlying replication mechanisms do not limit users to a single replication model. Applications can implement primary site, dynamic and shared ownership models as well as fail-over configurations.

Primary Site Ownership

With primary site ownership, asynchronously replicated data is "owned" by one site. Ownership means that the site may update the data. Other sites "subscribe" to the data owned by the primary site, which means that they have access to read-only copies on their local system. Primary site replication has many uses. These include:

- **Distribution of centralized information.** For example, product information such as price lists can be maintained at a corporate headquarters site and replicated to read-only copies maintained on order entry systems at remote sales offices.
- **Consolidation of remote information.** For example, inventory data maintained on systems in a number of remote warehouse locations can be replicated to a consolidated read-only copy of the data at a corporate headquarters site.

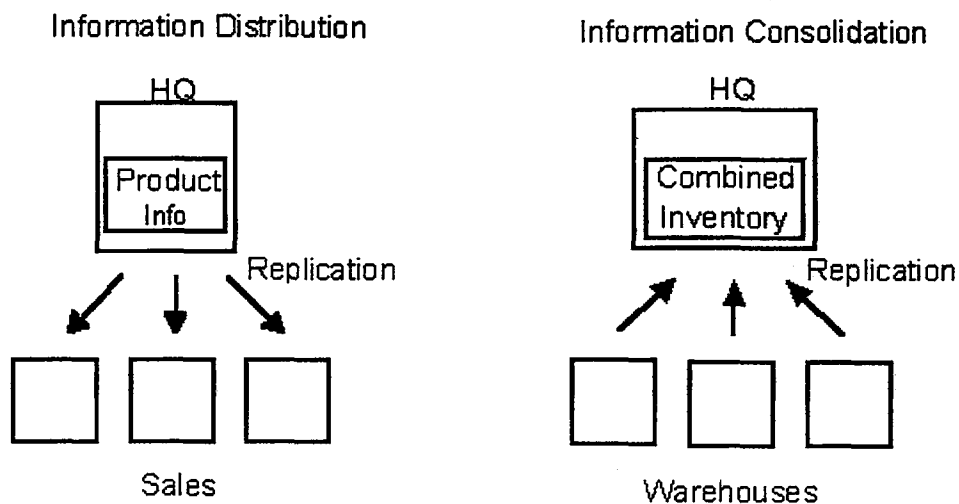


Figure 5.2 Two different uses of Primary Site Ownership

By restricting updates to certain sites, primary site ownership avoids conflicts. A primary site may own the data in an entire table in which case other sites subscribe to read-only copies of all or some subset of that table. Alternatively, multiple sites may own distinct subsets or partitions of the table. Each site might own a distinct set of rows, i.e., a horizontal partition, or a distinct set of columns, i.e., a vertical partition, within a table. Other sites then subscribe to read-only copies of all or some further subsets of the partitions.

Dynamic Ownership

With dynamic ownership the ability or right to update asynchronously replicated data moves from site to site while ensuring that at any given point in time only one site may update the data. For example, within an order processing system the processing of orders typically follows a well ordered series of steps, e.g., orders are entered, approved, shipped, billed, collected, accounted for, etc. Centralized systems allow the application modules that perform these steps to act on the same data contained in one integrated database. Each application module acts on an order, i.e., performs updates to the order data, when the state of the order indicates that the previous processing

steps have been completed. For example, the application module that ships an order will do so only after the order has been entered and approved.

By employing a dynamic ownership replication methodology such a system can be distributed across multiple sites and databases. Application modules can reside on different systems. For example, order entry and approval can be performed on one system, shipping on another, billing on another, and so on. Order data is replicated to a site when its state indicates that it is ready for the processing step performed by that site. Data may also be replicated to sites that need read-only access to the data. For example, order entry sites may wish to monitor the progression of processing steps for the orders they enter.

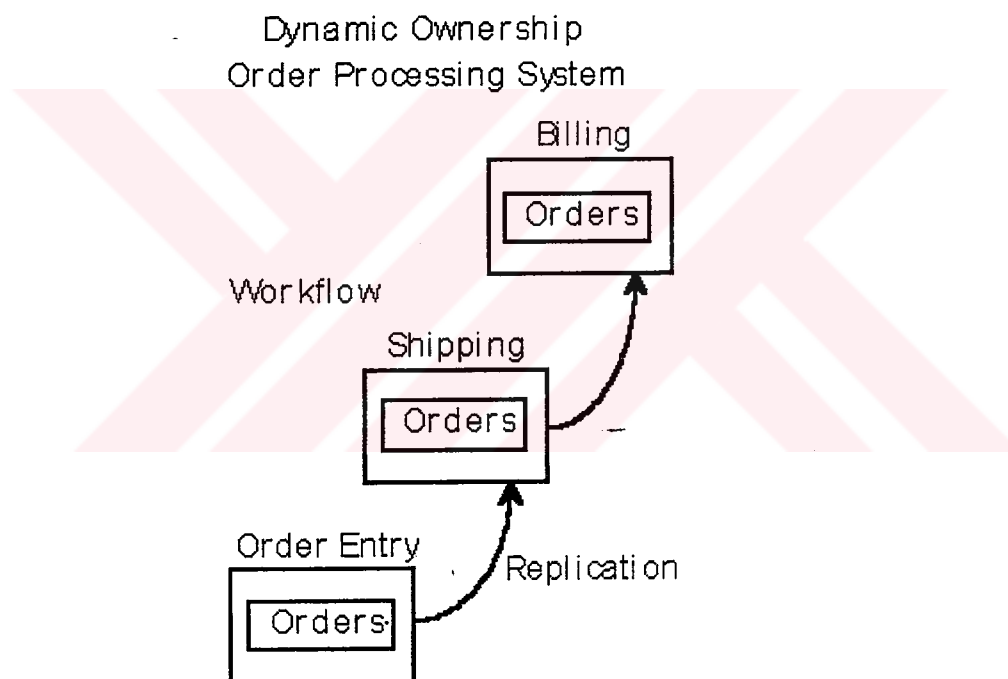


Figure 5.3 Dynamic Ownership

Shared Ownership

All of the replication models described thus far, i.e., primary site and dynamic ownership, share a common property; at any given point in time only one site may update the data while the other sites have read-only access to replicated copies of the data. In some situations, however, it is desirable to allow multiple sites to update the same data, potentially at the same time. For example, it may be desirable to replicate customer data across multiple sites and systems rather than maintaining customer data centrally. Different site, though, may need to update this data. Shared ownership allows asynchronous replication to be employed where primary site and dynamic ownership models would be too restrictive. As such, in those cases where temporary inconsistencies can be permitted, but in this case, conflict detection and resolution methods should be employed.

For example, earlier we discussed how a distributed order entry system could be implemented using primary site replication methodologies with horizontal partitioning. In this scenario each sales office owned a distinct horizontal partition of the tables containing orders and customer information for the customers serviced by each office. Each sales office entered orders for its customers, but no others. For some businesses, though, this is not the model. For example, a retail chain may have several stores in a metropolitan area. Customers may frequent the store closest to where they live but they will go into other stores and these other stores will want to take their orders when they do. If multiple stores perform updates to the same customer and order data, however, update conflicts potentially could occur. Sophisticated application developers can identify these conflicts and either select standard resolution routines or devise their own to implement such systems.

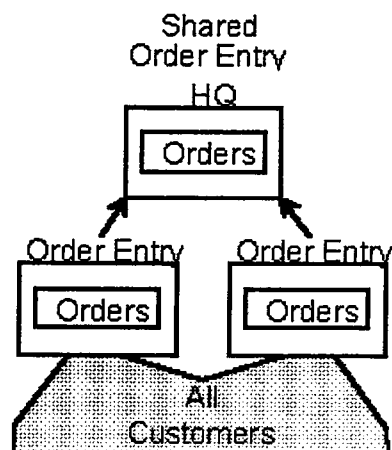


Figure 5.4 Shared Ownership

- **Full Transactional Consistency** The referential integrity of replicated data is ensured.
- **Automatic Update Conflict Detection and Resolution** Resolution routines can be selected declaratively from a set of predefined standard routines such as most recent timestamp or site priority. Users can also define their own customized resolution rules.
- **Full and Subset Table Replication** All rows or only selected rows in a table can be replicated. Two mechanisms are supported: multiple masters and updatable snapshots. These two mechanisms can be combined in hybrid configurations to meet different needs.

Multiple Masters

Multiple master replication supports full table, peer-to-peer replication between master tables. All master tables at all sites can potentially be updated depending on the replication model being employed. Changes applied to any master tables are propagated and applied directly to all other master tables. Failures of any one site containing master replicated tables will not block propagation of changes between other master sites.

Multiple master replication uses deferred RPCs (described below) as the underlying transport mechanism to propagate and apply changes. Changes to multiple master tables are applied in a transactionally consistent manner to ensure data and referential consistency. Changes are propagated either immediately, i.e., in an event-based manner, or at specified points in time when connectivity is available or when communications costs are lowest, e.g., during evening hours. If a remote system is unavailable, the deferred RPCs propagating change to that system remain in their local queue for later execution.

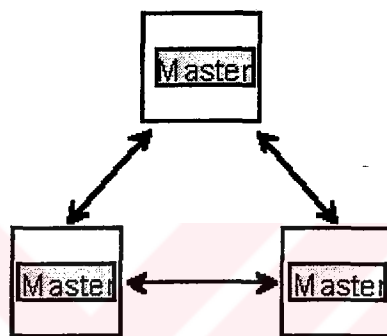


Figure 5.5 Multiple Master Replication

Updatable Snapshots

Oracle has extended the initial Oracle7 snapshot mechanism to support Symmetric Replication. Snapshots, as well as the snapshot masters, can be updated. Updates to snapshots are propagated and applied to snapshot masters using deferred RPCs as the underlying mechanism.

Snapshots can be defined to contain a full copy of a master table or a defined subset of the rows in the master table that satisfy a value-based selection criterion. Snapshots are refreshed from the master at time-based intervals or on demand. Any changes to the master table since the last refresh are then propagated and applied to the snapshot. Multiple snapshots are

refreshed from multiple masters in a transactionally consistent manner to ensure data and referential integrity.

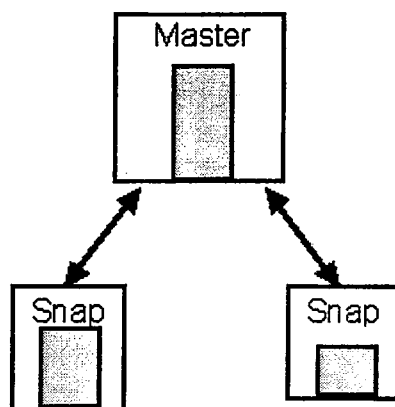


Figure 5.6 Updatable Snapshots

Hybrid Configurations

Multiple master replication and updatable snapshots can be combined in hybrid configurations to meet different needs (Figure 5.7). Specifically, snapshot masters can be replicated in multiple master configurations. This allows full-table and table subset replication to be combined in one system.

For example, multiple master replication between two snapshot masters can support replication between two hub sites supporting two geographic regions. Snapshots can be defined on the masters to replicate full tables or table subsets to sites within each region. This configuration allows the two master sites to function as fail-over sites for each other. An added benefit of this configuration is that snapshots can be re-mastered from the other hub site to provide an added measure of high availability.

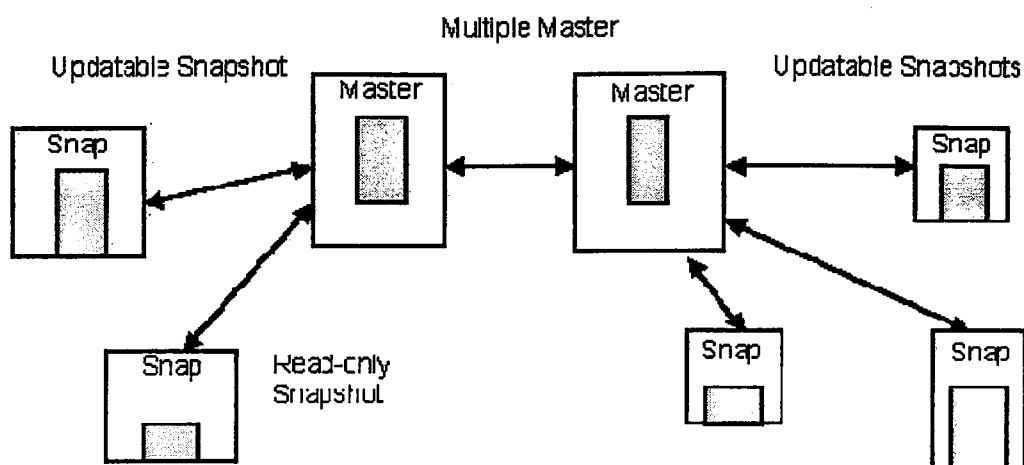


Figure 5.7 Hybrid Configuration

- **Event and Demand-Based Replication Methods** Replicated row changes can be efficiently propagated either immediately or when they are demanded by the target system.
- **Deferred Remote Procedure Calls (RPCs)** Remote PL/SQL procedures can be executed in an asynchronous, or store-and-forward, manner.

Deferred RPCs are a flexible, general purpose facility. They are used as a propagation mechanism for replication. The facility is also available for direct use to enable calls to remote PL/SQL procedures to be processed in an asynchronous or store-and-forward manner.

A local transaction initiates the execution of a deferred RPC by submitting a request to a local propagation queue. Submission into the queue is done within the local transaction. Entries in the queue are then pushed to their target location(s) and executed as a second step within separate transactions. If a remote system is unavailable when the deferred RPC queue is pushed, the entries for that target system remain in the queue for later propagation. The deferred RPC queue is durable, protected by the backup and recovery mechanisms of the Oracle7 server. This guarantees that the request will never

be lost and can be propagated and executed when the target system becomes available.

Deferred RPCs can be easily targeted to one or multiple remote systems. Multiple deferred RPCs submitted with the same local transaction are executed together within the same transaction remotely.

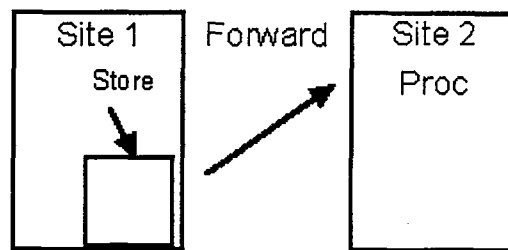


Figure 5.8 Deferred RPCs

Symmetric Replication generates replication support automatically. No coding is required. Oracle provides powerful management tools integrated into the database including:

- **Replication Catalog** Provides a single, consolidated repository of meta data that defines the distributed / replicated environment, i.e., what database objects (tables, procedures, triggers, indexes, etc.) are replicated where and how they are being replicated. The replication catalog is itself replicated to multiple sites to ensure high availability and easy local access to authorized users.
- **Distributed Schema Management** Allows replicated environments to be defined and changed automatically at multiple site by replicating and applying data definition language (DDL) commands. For example, operations such as adding an index or check constraint to a table everywhere it is replicated can be done automatically without requiring tedious and error-prone manual operations.
- **Server Manager** New Server Manager extensions provide an easy to use, GUI-based administration capability for Symmetric Replication. Allows

administrators to easily query the replication catalog, examine internal replication engine components, initiate distributed schema management operations, and troubleshoot problems.

Symmetric Replication is built into the Oracle7 server as an internal, integrated facility.

- **Single Server** No additional, external components are required that must be configured, monitored and maintained.
- **Standard Components** Symmetric Replication is implemented using proven database components such as tables, views, and PL/SQL procedures that Oracle users are already familiar with.
- **Standard Backup/Recovery** Symmetric Replication is protected by Oracle7's standard backup and recovery mechanisms and procedures. No additional complex procedures involving extra external components are required. Database systems can be backed-up and recovered separately without the need to synchronize operations with other sites.

5.1.2. Sybase

The information below is taken from [Sybase1] and [Sybase2].

As workforces become increasingly decentralized, managing information effectively is perhaps the single greatest challenge for businesses today. Sybase, the pioneer in client/server databases, offers a complete solution of database, middleware, tools, and services that help users access information where and when they need it. Sybase tightly integrates these software categories -database, middleware and tools- into a single architectural framework that supplies products optimized specifically for the task at hand.

Database solution

When businesses need outstanding performance and scalability, Sybase offers the Sybase SQL Server database family. For extremely large databases, Sybase MPP for massively parallel processing is an optional extension to SQL Server, as is Sybase IQ, designed for interactive query processing in the data warehouse. SQL Anywhere supports mass deployment of SQL databases.

Middleware solution

The Enterprise CONNECT family of products is the industry leader in interoperability. With Enterprise CONNECT, multiple data sources plug and play together, and any data source can connect to any development tool. Sybase's middleware products include Replication Server, OmniCONNECT, MDI Database Gateway, and Enterprise Messaging Services.

Tools

PowerBuilder from Powersoft, a division of Sybase, is the standard for developing client/server applications that access corporate data. Other Powersoft tools include InfoMaker for personal data access and reporting, S-Designer for powerful database design, and Watcom compilers for efficient C/C++ development.

Replication Server

A reliable replication system must do much more than simply copy a piece of data. The system must also:

- maintain the integrity of the data at the transaction level,
- deliver data quickly and efficiently across the network,
- allow distributed sites to modify data,
- be easy to monitor and manage (the most important, perhaps),
- transfer data in any direction across heterogeneous data sources.

Two-phase commit protocol is too expensive for normal operations and can stop the entire system in the face of any component failure. Other approaches use simplistic table snapshot mechanisms, which are not transaction based, so copies lack basic relational integrity. Sybase Replication Server is the product which claims providing the real solution to these problems.

Depending on the design of the distributed system, which uses Replication Server, there are three types of sites in a distributed system:

- Primary (or sites with primary data only): Locations from which data is replicated to other systems. Primary data can be modified on these sites.
- Secondary (or sites with replicate data only): Locations, which receive data from one or more primary sites. The replicated data is read only and cannot be modified.
- Peer-to-Peer (or hybrid sites with primary and replicate data): Locations from which data is replicated to other systems, and which receive data from one or more other primary sites. Peer-to-peer replication environments may involve applications where update conflicts are avoided or applications where update conflicts are resolved.

Sybase Replication Server may be used for applications that include any combination of these types of sites. Information may be replicated to and from Sybase or non-Sybase data sources.

Replication Server synchronizes replicate copies of data on heterogeneous platforms throughout a client/server network. It furnishes the highest on-line transaction processing (OLTP) performance, while enabling you to eliminate the inherent conflict between OLTP and decision-support performance. It also enables a high degree of local autonomy and flexibility. Remote sites can update replicate data by using a safe remote update model. By transforming information in flight, Replication Server synchronizes databases with different schemas, formats, and

naming conventions. Flexible event replication allows remote notification of events, so each site can respond based on local needs.

The Replication Server design supports "fail-through" computing - an applications architecture that allows users to continue their work even though system components are unavailable. It has reliable store and forward capability. If a remote site is accessible, the information is forwarded to that site; if one or more remote sites become unavailable, the information is stored until the connections are re-established, at which point copies are automatically resynchronized. Corporate sites can continue operations using their local copy of the data even when remote sites are inaccessible.

Replication Server solves several major customer problems:

- corporate consolidation: Replication Server lets you maintain a corporate overview of distributed operations that is very close to real time, even when distributed business units run on a variety of hardware and DBMS platforms
- decentralization: Replication Server allows you to locate data where it is needed, making distributed business units much less susceptible to central computer or network downtime while reducing overall communication costs. It also allows bidirectional data sharing with a safe approach for replicating remote updates
- high availability: Replication Server systems remain robust despite typical hardware, software, and network failures, delivering applications with very high uptime at a reasonable cost
- live decision support: Replication Server can replicate an OLTP database, allowing your analysts to run complex decision-support queries on data that is within seconds of real time, without affecting OLTP performance
- disaster recovery: Replication Server lets you maintain a near-real-time "warm standby" database to which applications can switch with virtually no downtime if the primary site fails

With the prevalence of heterogeneous database environments, many organizations require two-way replication between multivendor sources and targets. Replication Server, along with the replication agents for different DBMSs, provides a reliable and maintainable solution that improved reconciliation between systems, reduced operational and system costs, and facilitated the conversion. It leverages Sybase Enterprise CONNECT technology to integrate multivendor targets. Enterprise CONNECT gateways use the same Open Server interfaces as Replication Server, so it is easy to combine these components. A special Replication Driver for ODBC allows the system to distribute information to a wide range of ODBC compliant databases. Together, Replication Agent, Enterprise CONNECT gateways, and the Replication Driver for ODBC allow to integrate platforms ranging from laptops to desktops to mainframes.

Sybase Replication Server meets all the critical business needs including the delivery of live information to remote locations, corporate consolidation of information from autonomous units, real-time decision support, and continuous operations despite unexpected outages. Sybase Replication Server fulfils these needs in a fully heterogeneous environment and allows a high degree of local customization. Across any number of locations, it replicates data to each local processing site. Its event-driven replication and reliable store-and-forward mechanism ensure that remote locations have information that is as close to real time as possible. It can work across heterogeneous data sources, pulling together all data scattered across an organization. While each branch maintains control over its own primary data, some or all data is replicated to corporate headquarters, so users throughout the organization can view up-to-date corporate information at all times.

Because Replication Server replicates data to local processing sites, failures on networks and remote systems rarely matter to users. Sites can continue operation on their local copy of the data when the primary data source is down. Many organizations use Replication Server to create an alternate primary or a "warm standby" system. Even when the primary data site fails for an extended period, users continue to get the information they need because the system can fail over to an

alternate primary. If a remote site fails, it will be automatically resynchronized with the primary data source when it comes back on line.

Technical Highlights

- transaction replication maintains the transactional integrity of all distributed information
- business rules, when enforced enterprise wide, ensure quality and consistency of data
- primary sites retain control over their data
- Replication Server copies to and from heterogeneous databases. Replication Agents integrate multivendor sources. Enterprise CONNECT gateways and the Replication Driver for ODBC integrate multivendor targets
- data relocation is transparent
- data is accessible locally and transparently, ensuring ease of access for all users
- easy-to-use graphical user interfaces give you access to component availability, transaction tracking and routing, and security management
- sites specify the data they need, down to the row, then subscribe to it to receive regular updates of critical information
- Replication Server detects updates at the primary site
- Replication Server asynchronously delivers complete transactions to maintain up-to-date information at all subscribed sites
- user-specified in-flight transformations allow the system to replicate data between sites despite schema and datatype differences
- users have a wide range of options for specifying subscriptions: a SQL-like subscription facility allows users to specify conditions; dynamic transaction routing enables applications to specify the routing of information at run time; subscriptions to stored procedure parameters notify remote sites of specified business events

5.1.3. PeerDirect

The information below is taken from [Rennhackkamp, 1998]. First examining all the prospects of PeerDirect, in the evaluation section, Martin Rennhackkamp evaluates this product according to his criteria for a good replicator which is being transaction-based, serializable, asynchronous, unobtrusive, robust, configurable, manageable, efficient, and transparent. Then in section “Replication Evaluation Checklist”, a list of the key issues for a replication product prepared by PeerDirect Inc. will be given.

Design Issues

With PeerDirect, a database is maintained at a site. The database is considered as a collection of work sets. A work set is a set of closely related tables, such as customers and customer details, or invoices and invoice lines. A work set is further divided into slices, each of which represents an instance of the work set. For example, a particular customer, with its details, is one slice. A base record in a base table uniquely identifies each slice. For example, each customer row in the customers table is a base record. The choice of base table depends on your business rules. The only thing that PeerDirect requires is that each work set has a base table. A work set can be nested under another work set, and the nested tables are then replicated with the parent work set.

One of PeerDirect’s design goals is that each site stores all and only the data the users require. Users who have been granted the `da_subscribe` right by the initial admin user must subscribe to the slices in which they (and other users at the site) are interested. The users with `da_subscribe` rights at any site can subscribe to any grouping of slices. For this purpose, all the nonnested base records are replicated to each site so that users can select the records of interest. Their sites will only store and maintain the data of the nested rows to which they subscribe. When the users with the `da_subscribe` rights unsubscribe from a slice, the slice’s data is deleted from the local site. However, it is still maintained at all the other subscribing sites. It is

important to unsubscribe from a slice only after all operations on the slice have been successfully replicated. A table can be designated as global, which means it will be replicated in its entirety to every site. The "admin" user, who is automatically created when PeerDirect is loaded at the first site, can create new users and grant and revoke rights to them.

Furthermore, table columns can be grouped in "fragments," or sets of nonprimary-key columns that are updated together. PeerDirect replicates only the updated fragments, rather than the entire record, to the interested sites. In some scenarios this replication can eliminate update collisions. It can also reduce network traffic and replication processing loads. However, PeerDirect requires each fragment to have a stamp column, which is a 26-character field, whose name must begin with the letters stamp. PeerDirect uses the stamp column to store the ID of the user who last updated the fragment, the site and time at which this update was made, and an encrypted integrity check value. PeerDirect automatically adds a stamp column to each replicated table, but if you want to use fragmentation, you must add a stamp column for each fragment. Because the primary key is considered a nonupdatable part of each fragment, its values cannot be changed -you must delete and insert rows to change the primary key values in a fragmentation scheme. PeerDirect supports key columns of byte, short, long, string, Boolean, and timestamp datatypes.

When you "PeerDirect-enable" a database (as they phrase it), you run a utility that creates several system tables in the database, which are used to maintain the replication configuration. The names of these tables are all shorter than eight characters, and they all start with "d," as in dSite, dUsr, dMsg, and so on. Application programs should not access these tables, and your database design should not contain similarly named tables.

Distribution and Subscription

It is difficult with any replication system to define the entire data replication scheme. With PeerDirect, you use a C-like language called Distribution Control (DC)

to define how your databases should be replicated, shared, and secured. You use DC to define the work sets, fragments, encrypted columns, operations, and replication rules in script files. A DC script file can include other DC script files. The script is compiled using PeerDirect's DC compiler, DCC, which ensures that the script's rules match the database's physical structure. The distribution information is then considered part of the database schema and is stored in a DC file associated with the database. Scripts can be ported between systems easily, and they can be generated from CASE tools and other similar systems. GUI tools, on the other hand, often require to redefine the replication configuration interactively -and manually- against each source database. A GUI-based administration tool is planned for a subsequent release of PeerDirect.

Users can subscribe and unsubscribe to slices using the PeerDirect Administrator or by calling the PeerDirect APIs. With the PeerDirect Administrator, you can select base tables and subscribe to individual slices. However, the user must have the necessary administrative and subscription rights. Subscriptions, as well as subscription rights, can also be maintained through the PeerDirect API calls. The alternative to subscriptions is to designate a site as a Full Site, meaning it will contain a copy of the entire database. With PeerDirect 3.0, users can automatically subscribe to work sets. This ensures that their sites will regularly receive all the new data allocated to that work set on the source database. For example, if you have auto-subscribed to the customer work set, any new customers will automatically be replicated to you.

Projects

Before an application can access a database replicated by PeerDirect, it must be registered as a PeerDirect project. The registration occurs against the development database. Each project's unique name is used to add information about the application to the PeerDirect development environment. This includes the "organization" (department or team) developing the project, optionally with a password. For each project you must also register editions and networks. You can

use the PeerDirect Administrator for these tasks, or you can use command-line utilities. Ideally, a replicator only needs to focus on changes to the database contents, not necessarily the applications that access it and from this ideal point of view, this can be a shortcoming of the PeerDirect replicator.

Configurations

PeerDirect can be implemented in one of many replication configurations. It currently supports Oracle7 and 8, Microsoft SQL Server, Informix, Sybase SQL Anywhere (with plans for Adaptive Server Anywhere), Microsoft Access, and Corel Paradox databases on any platform, with support for Sybase SQL Server, IBM DB2, Pervasive, and others planned for later in 1998. The PeerDirect Replication Server runs on Windows 95 and Windows NT, and it accesses the database as a client application, so the database itself can reside on a Windows, Unix, or mainframe server.

PeerDirect does not force you into a particular replication configuration or data allocation scheme because of replication software or available platform limitations. It supports bidirectional peer-to-peer replication among heterogeneous databases without requiring that you nominate master and slave sites.

Deletions

Delete operations are a bit more complex than insert and update operations when using PeerDirect replication. When PeerDirect detects a deleted row, it assumes the row was deleted accidentally and tries to restore the row from another site. As a result, you must delete rows using the PeerDirect API (calling the procedure DSECDeleteRecord) or by inserting rows in the PeerDirect system table (called dDel). When you call the DSECDeleteRecord procedure, PeerDirect ensures that all related records are also deleted. With the system table approach, you write the key of the row to be deleted in the dDel system table using a specified syntax. This can be

done through a trigger, for example. During the next replication cycle, PeerDirect calls DSECDelRecord for every entry in the dDel table.

Transactions

PeerDirect 3.0 does not support transactions. It replicates the operations performed on the work sets and fragments.

Evaluation of Martin Rennhackkamp

Considering the criteria for a good replicator, namely transaction-based, serializable, asynchronous, unobtrusive, robust, configurable, manageable, efficient, and transparent, PeerDirect fares pretty well. It replicates asynchronously, it is robust, and it can handle different types of problems. It is highly configurable. It can replicate peer-to-peer, update-anywhere operations in many directions, which some competing products cannot do. It can replicate fragments, which few other replicators can do. It is easy to manage, especially when scaling up to a large number of databases. Most important, it can replicate transparently between Oracle7 and 8, Microsoft SQL Server, Informix, Sybase SQL Anywhere, Microsoft Access, and Corel Paradox databases, which very few other replication products can do.

However, in my opinion PeerDirect has two problem areas. First, it doesn't replicate within transaction boundaries, which means that you can have the effects of partial transactions replicated to some sites. This problem should be addressed in the subsequent release. Second, PeerDirect 3.0's implementation of delete logic is not always unobtrusive. Some applications may require PeerDirect API calls to delete rows, while other databases may require additional triggers to implement its delete procedures. You may also argue that the requirement to add additional stamp columns for fragmentation is not unobtrusive, but you must agree that fragmentation is very difficult to implement, and few replicators even attempt it. The price of these database changes for fragmentation may not be so high if you need that kind of logic.

Finally — and this is true of any replication product — a replicator is a tool you use to implement a business solution. The replicator itself is not the business solution. You still have to design and implement a replication scheme, which in some cases may be pretty complex. The replication product only gives you the capabilities to put your business solution into action. It can influence how well your business solution is implemented, but you first have to design the business solution.

Replication Evaluation Checklist

When evaluating a replication product, these are key issues to watch:

- Open System - Heterogeneous, Standards-Based
- Flexibility and Scalability
- Performance and Efficiency
- Integrity and Reliability
- Security
- Ease of Use - For Developers
- Ease of Use - For Runtime Administration
- Ease of Use - For End Users

5.1.4. IBM Solution – DRDA

The information below is taken from [IBM_DRDA].

DRDA (Distributed Relational Database Architecture) is a technology that enables reliable, secure, high performance data access for integrated client/server solutions. It is an architecture for distributed database protocols that any software vendor can use to develop connectivity solutions that provide applications with access to remote data. DRDA defines what information must be exchanged and how it must be exchanged, enabling coordinated communications between systems. DRDA provides full client/server facilities across heterogeneous systems and

databases and is part of IBM's Open Blueprint , the market-leading approach for open distributed computing.

Applications developers benefit from DRDA connectivity because they continue to use the standard language for data access: SQL, the Structured Query Language, in either its embedded or callable form. DRDA defines industry-leading technology for database interoperability:

- Remote Unit of Work - access to a single remote database for processing related SQL statements
- Distributed Unit of Work - access to multiple remote databases with the ability to commit updates across all of them
- Stored Procedures - invocation of procedures at the remote database location
- DCE Security - identification and authentication of users in environments using the Distributed Computing Environment (DCE) from the Open Software Foundation (OSF).

IBM products for end user access, data replication, systems management, and application development are all enabled by DRDA connections. Database applications or tools that are coded using standard SQL APIs (i.e., embedded SQL, SQL CLI or ODBC) can automatically access remote databases merely because the underlying database client code handling the SQL is DRDA-enabled, freeing the application from needing to know the location of the database in the network and from dealing with the communications code to access it.

Many different software companies have licensed DRDA. Attachmate Corp., File Tek. Inc., GrandView DB/DC Systems, Informix Software, Inc., Object Technology Int'l., Oracle Corp., Rocket Software, Inc., StarQuest Software, Inc., Sybase/MDI, Wall Data, Inc., and XDB Systems, Inc. are companies that have products announced or available that implement DRDA.

5.1.5. Mariposa Distributed Database Management System

The Mariposa distributed database management system is an ongoing research project at the University of California at Berkeley. An information below is taken from [MARIPOSA_manual] and [Sidell et al., 1996].

Mariposa addresses fundamental problems in the standard approach to distributed data management. Mariposa allows DBMS's which are far apart and under different administrative domains to work together to process queries. Furthermore, it introduced an economic paradigm in which processing sites buy and sell data and query processing services. A Mariposa system consists of a collection of sites, which provide storage, query processing and name service. Mariposa has been designed with the following principles:

- Scalability to a large number of cooperating sites. In a WAN environment, there may be a large number of sites. The goal is to scale to 10,000 servers.
- Local autonomy. Each site must have control over its resources. This includes which objects to store and which queries to run. Query and data allocation cannot be done by a central, authoritarian query optimizer.
- Data mobility. It should be easy and efficient to change the “home” of an object. Preferably, the object should remain available during movement.
- No global synchronization. Updates and schema changes should not force a site to synchronize with all other sites. Otherwise, many common operations will have exceptionally poor response time.
- Easily configurable policies. It should be easy for a local database administrator to change the behavior of a Mariposa site. A Mariposa system should respond gracefully to changes in user activity and data access patterns to maintain low response time and high system throughput.

Overview of the Architecture

When a query is generated at a site, which is, called a “home site”, it is sent from the frontend application to the Mariposa program running on the server in home site. The query is passed through a **parser**, which checks for syntactic correctness and performs type checking; an **optimizer**, which produces a query plan that describes the order in which different steps in the plan will be executed; and a **fragmenter**, which changes the plan produced by the optimizer to reflect the data fragmentation (Every Mariposa table -class- is horizontally partitioned into a collection of fragments which together store the instances of the table). The final result produced by the fragmenter is the fragmented query plan. In order to do their work, the parser, optimizer and fragmenter need information about data types, fragment location, etc. This information is maintained by a **Mariposa name server**. The fragmented query plan describes the operations that will be performed in order to execute the query, and the order in which they will be carried out. The fragmented query plan is passed to the **query broker**, whose job it is to decide where each piece of the fragmented query plan will be executed. The query broker uses one of two protocols:

- In the long protocol, the query broker contacts the **bidder** module at each potential processing site. The bidder responds to requests for bids from the query broker. When a bidder receives a request to bid on part of a query, it may either refuse to bid, or return a bid to the query broker. The bid includes the price to perform the work, and a time bound within which the work must be completed. If a bidder bids, then it must process the query if it is chosen by the query broker to do so. The broker waits for responses from the bidders before selecting the best ones.
- In the short protocol, the query broker uses information collected from the name server to decide which sites will process the query. It does not contact the processing sites.

After the query broker has specified the processing sites, the backend's **coordinator** module takes over. The coordinator notifies the remote sites to begin processing, collects the results, and returns the answer to the client program.

The Mariposa Replica System

Mariposa permits the replication of data fragments. In the current implementation, a replica is created from one other replica, which is referred to as its parent. Replicas created from a parent are called its children. Each replica periodically receives updates from its parent. This allows Mariposa to use one replica in the place of another, improving availability during host crashes and network failures, and improving performance. There are two types of Mariposa replicas: A read-only replica receives all updates from its parent but cannot process updates; a read-write replica propagates its updates to its children, as well as receiving updates from its parent, if it has one. The term update is used to mean any tuple insertion, deletion, or modification.

Acquiring and maintaining a copy may be thought of as applying streams of updates from other copy holders, with associated processing costs. In the economic parlance of Mariposa, a site buys a copy from another site and negotiates to pay for update streams. The process of buying a new copy is relatively simple. Assume a Mariposa site S1 owns the only copy of fragment F. If another site, S2, wishes to store a copy of F, then S2 must perform the following steps:

1. Negotiate for updates: S2 negotiates with S1 to buy an update stream for F. This contract specifies an update interval T and an update price P . The update interval specifies that writes to F at S1 will be forwarded to S2 within T time units of transaction commitment at S1. An update stream contains changes only from committed transactions. In this way, S2 can be assured that its copy of F is out of date by an amount bounded by T plus the maximum network delay plus the maximum time to install the update stream. In return for the update stream, S2 will pay S1 P dollars every T time units.

2. Negotiate reverse updates: If S2 wants to write to its copy of F, then it must also contract with S1 to accept an update stream generated at S2. In this case, there are two update intervals: $T_{1 \Rightarrow 2}$ and $T_{2 \Rightarrow 1}$, which are not necessarily the same. $T_{1 \Rightarrow 2}$ is the frequency with which S1 updates S2, and $T_{2 \Rightarrow 1}$ is the converse. In this case, the price P mentioned in step (1) above is the price paid by S2 to S1 for S1 sending updates to S2, and for S1 receiving updates from S2.
3. Construct an initial copy: S2 contracts with S1 to run the query `SELECT * FROM F`. S2 will install the result of this query and begin to apply the update stream generated by S1. If S2 is writing its copy of F, it starts to send updates to S1 as well.

If a site no longer wishes to maintain a copy, it has a number of options:

- Drop its copy. That is, stop paying for its update streams, delete the fragment and stop bidding on queries involving the fragment.
- Sell the copy. The site can try to sell its update streams to someone else, presumably at a profit. If so, then the seller must inform all the creators of update streams to redirect them to the buyer.
- Stop updating, that is, stop paying for its update streams but don't delete the fragment. The fragment will become more and more out of date as updates are made at other sites. If the fragment is split or coalesced, the fragment will essentially become a view. This view is unlikely to be very useful, since it is unlikely that queries over the relation will correspond exactly to the view. Therefore, doing nothing is a possible but not very effective action.

In Mariposa, one copy of each fragment is designed to be the master copy. All other copies can be freely deleted, but the master copy must be retained until sold. This will prevent the last copy of a fragment from being deleted. In addition, the notion of a master fragment is needed to support systematic splitting and coalescing of fragments.

Although Mariposa can provide standard consistency guarantees two-phase commit, it also supports temporal divergence between replicas. Mariposa provides consistency at the cost of greater staleness.

Mariposa Name Service

The purpose of name service is to supply client sites with the necessary information to run queries on remote data. In order to process a query on remote data, Mariposa needs the queried tables' metadata at various stages:

- During parsing, the syntactic correctness of the query statement has to be verified. This requires information about the queried tables' attributes and their types, about operators used in the query etc.
- The fragmenter needs information about the fragmentation of remote tables.
- The query broker needs to know the location of the remote frgements.

For local tables, this information is stored in the site's local database catalogs. For remote tables, a name server provides the information stored in the remote database catalogs to its clients by replicating the remote catalogs. A Mariposa name server is a regular Mariposa site, which keeps read-only copies of a subset of the system catalogs of other sites. These copies are maintained by the update streams sent from the source sites to the name server; as a consequence, information obtained from a name server will always be out-of-date by a certain amount. The set of sites whose system tables are on a name server is not fixed and does not have to include every existing site. The DBA of a site autonomously determines if that site should also provide name service and which remote sites' catalogs it should replicate.

5.2. Unsolved problems

Although all different solutions of different products existing today cover important aspects of distributed database technology, there are still significant

research problems that remain to be solved. The problems below are some important ones taken from [Casavant & Singhal, 1994] and summarized:

- Since there is no full understanding of the performance implications in distributed database design, there is a problem in adopting current protocols and algorithms to distributed database systems as the systems become geographically distributed or as the number of system components increases. There are plenty of performance studies of distributed DBMS, and these usually employ simplistic models, artificial work loads, or conflicting assumptions, or they consider only a few special algorithms. The proper way to deal with scalability issues is to develop general and sufficiently powerful performance models, measurement tools, and methodologies.
- There is no underlying design methodology that combines the horizontal and vertical partitioning techniques; horizontal and vertical partitioning algorithms have been developed completely independently. What is needed is a distribution design methodology that encompasses the horizontal and vertical fragmentation algorithms and uses them as part of a more general strategy. Such a methodology should take a global relation together with a set of design criteria and come up with a set of fragments, some of which are obtained via horizontal fragmentation, while others are obtained via vertical fragmentation. Another shortcoming of design algorithms is their simplification in the design step by isolating fragmentation and allocation steps. In fact both steps have similar inputs, differing only in that fragmentation works on global relations, whereas allocation considers the fragment relations. They both require information about the user applications (such as how often they access data and what the relationship of individual data objects to one another is), but they both ignore how each makes use of these inputs. What would be more promising is to extend a methodology discussed above so that the interdependence of the fragmentation and allocation decisions is properly reflected.
- Even though query languages are becoming increasingly powerful (for example, new versions of SQL), global query optimization typically focuses on a subset of the query language-namely, select-project-join (SPJ) queries with

conjunctive predicates. However there are other important queries that warrant optimization, such as queries with disjunctions, unions, aggregations, or sorting. Although these have been partially addressed in centralized query optimization, the solutions cannot always be extended to distributed execution. Another problem is with the query optimization cost. There is a necessary trade-off between optimization cost and quality of the generated execution plans. The optimization cost is mainly incurred by searching the solution space for alternative execution plans. Thus, there are two important components that affect optimization cost: the size of the solution space and the efficiency of the search strategy. In a distributed system, the solution space can be quite large, because of the wide range of distributed execution strategies. Therefore, it is critical to study the application of efficient search strategies that avoid the exhaustive search approach. More important, a different search strategy should be used depending on the kind of query (simple versus complex) and the application requirements (ad hoc versus repetitive).

- The field of data replication needs further experimentation, research on replication methods for computation and communication, and more work to enable the systematic exploitation of application-specific properties. Since there is not consistent framework for comparing competing techniques, it is difficult to compare different replication products. Another problem with replication is, all studies exist today were interested in data replication only, but there is a need for integrated systems in which the replication of data goes hand in hand with the replication of computation and communication (including I/O).
- As database technology enters new application domains, such as engineering design, software development, and office information systems, the nature and requirements for transactions change. These domains require transaction models that incorporate more abstract operations that execute on complex data. Thus, work is needed on more complicated transaction models and on correctness conditions different from serializability.
- From previous work, it is showed that running a DBMS as an ordinary application program on top of a host operating system is undesirable. Therefore

some ways are searched to integrate DBMS and operating system functions. Efforts that include too much of the database functionality inside the operating system kernel or those that modify tightly closed operating systems are proved to be unsuccessful. Because there is a mismatch between the requirements of the DBMS and the functionality of the existing OSs. This is even more true in the case of distributed DBMSs that require functions like distributed transaction support, including concurrency control and recovery; efficient management of distributed persistent data; and more complicated access methods that existing distributed OSs do not provide. The most logical way is, the operating system should implement only the essential OS services and those DBMS functions that it can efficiently implement and then should left the others to DBMS. The model that best fits this requirement seems to be the client-server architecture with a small kernel that provides the database functionality that can efficiently be provided and does not hinder the DBMS in efficiently implementing other services at the user level. However, which DBMS services can efficiently be implemented is a controversial issue. Naming, which is used to give transparent access to system resources; and transaction management are two debated issues on which there is not a common idea whether an OS can implement efficiently or not.

5.3. Future expectations

While trying to solve the problems which are unsolved in today's technology, unavoidably there will be new requirements and accordingly, this will results in the change of the existing solutions and development of new ones. By observing common trends in today's technology, we can forecast the trends of next-generation distributed DBMSs. These are changes expected in [Casavant & Singhal, 1994]:

- Advances in the development of cost-effective multiprocessors will make parallel database servers feasible. This will affect DDBSs in two ways. First, distributed DBMSs will be implemented on these parallel database servers, requiring the revision of most of the existing algorithms and protocols to

operate on the parallel machines. Second, the parallel database servers will be connected as servers to networks, requiring the development of distributed DBMSs that will have to deal with a hierarchy of data managers.

- As distributed database technology infiltrates nonbusiness data-processing-type application domains such as engineering databases, office information systems, and software development environments, the requirements of these systems will change and this will necessitate a shift in emphasis from relational systems to data models that are more powerful. Current research along these lines concentrates on object orientation and knowledge base systems.
- The diversity of distributed database application domains is probably not possible through a tightly integrated system design. Distributed database technology will need to effectively deal with environments that consist of a federation of autonomous systems. The requirement for interoperability between autonomous and possibly heterogeneous systems has prompted research in multidatabase systems.

CHAPTER SIX

INTRODUCTION TO AN INTELLIGENT INTERFACE

Under the light of all the information given in the previous chapters, an intelligent interface is trying to implement some of the functions of a distributed database management system. This interface is implemented as a distributed database model on the application layer and it is just simulating a distributed database engine supporting different databases connected through ODBC.

An intelligent interface can create a global view of a distributed database being designed according to these assumptions:

- All sites have DBMS's that can be different or the same (homogeneous or heterogeneous), but the conceptual schema of databases must be the same. Databases must be tightly integrated; meaning that they are not autonomous and all access to data must be through an intelligent interface. If all sites are autonomous, this means that, each site can use its DBMS without taking into account the global view of the system. The global view of the system will be given by our interface and if any DBMS executes some transactions directly, without using our interface, there will be inconsistencies and redundant data in the system. Therefore I think that, a tightly integrated distributed database system will be more suitable.
- Tables are fragmented according to primary and derived horizontal fragmentation (vertical fragmentation is not supported).

- Any table or any fragment of the tables can be replicated at more than one site. The intelligent interface uses asynchronous distributed technology; namely store and forward method for update propagation. In this method, transactions initiate operations which need to be propagated to other systems, but if these other systems are not available, the propagation will be deferred until the systems come back up. A queue mechanism is used for this purpose.
- The structure of tables, fragmentation, replication and allocation information, in other words all the information describing a distributed database is kept in a database, which can be called as a Name Server. An interface must be in connection with a name server to do a certain task.

6.1. The Structure of a Name Server Database

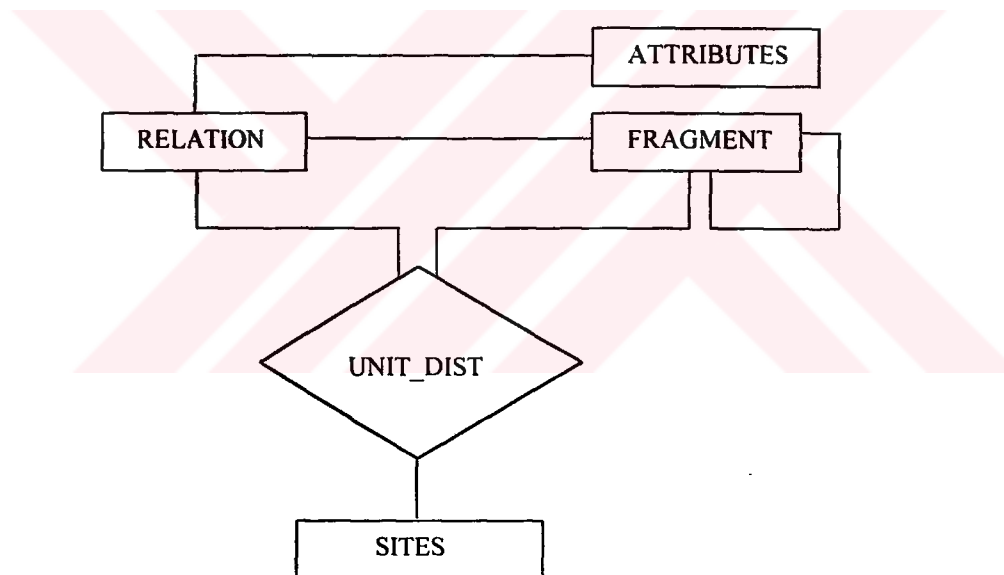


Figure 6.1 E²R Schema of the Name Server Database

The relation specific information is kept in a table called “RELATION”; the information about attributes of relations is kept in a table called “ATTRIBUTES”. Rel_id is a foreign key in the “ATTRIBUTES” table, to indicate the table to which an attribute belongs. As its name implies, the “FRAGMENT” table is keeping information about fragments. Since a fragment may belong to a table or a fragment,

an attribute called “parent”, indicates the table or the fragment to which a fragment belongs. A relation or a fragment, as a unit of distribution, can be replicated at many sites. This distribution information is kept in a relation called “UNIT_DIST”. Again as the name implies, “SITES” table keeps information about sites. Here the most important information is alias names of databases, since the connection to databases is established by using these aliases. It is obvious that the name server is the brain of our intelligent interface.

6.2. Query Processing

There are two main types of queries, data retrieval queries (selection), and data manipulation queries (insert, update, delete). The functions of the intelligent interface are implemented and tested on an example Student database. E²R Schema of this test database can be seen in Figure 6.2.

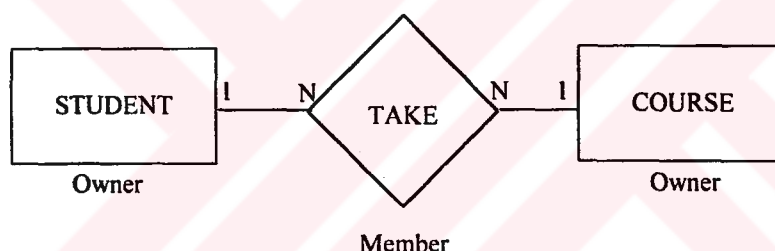


Figure 6.2 E²R Schema of an example Student Database

The schema seen in Figure 6.2 is the global conceptual schema from the user’s point of view. In this view, it is not possible to see any distribution information. Physically, the Student table is fragmented (since the Student table is the owner relation, Primary Horizontal Fragmentation is applied). Take table is fragmented according to the Student table (since it is a member relation, Derived Horizontal Fragmentation is applied). By allocating corresponding fragments into same sites, these fragments are distributed. Some fragments are replicated at other sites. Course table is entirely replicated at all sites. All this distribution information is recorded in the Name Server database and the task of an Intelligent Interface is giving

transparent access to the data by getting help from Name Server. To obtain integrated view of individual databases, global conceptual schema, fragmentation and allocation schema are needed. Fragmentation and allocation schemas of an example Student database can be seen in Figure 6.3 and 6.4.

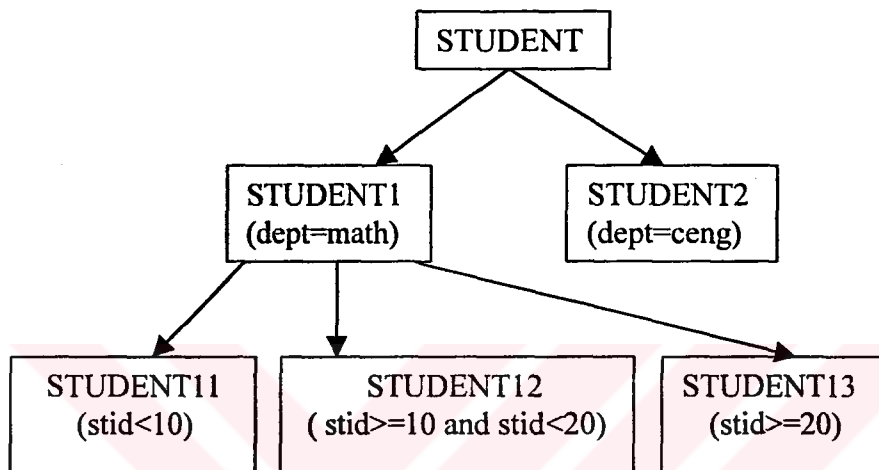


Figure 6.3.a Primary Horizontal Fragmentation on Student

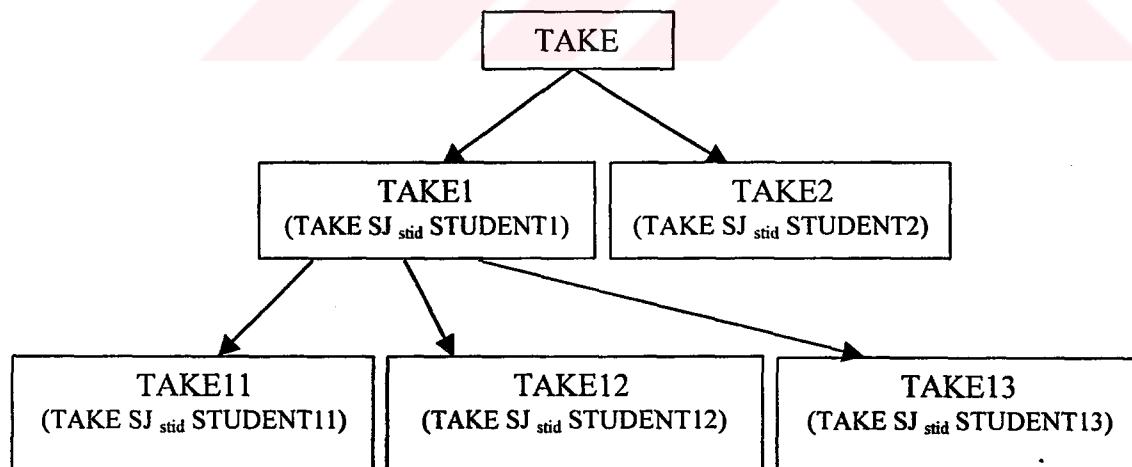


Figure 6.3.b Derived Horizontal Fragmentation on Take

Figure 6.3 Fragmentation Schema of an example Student Database

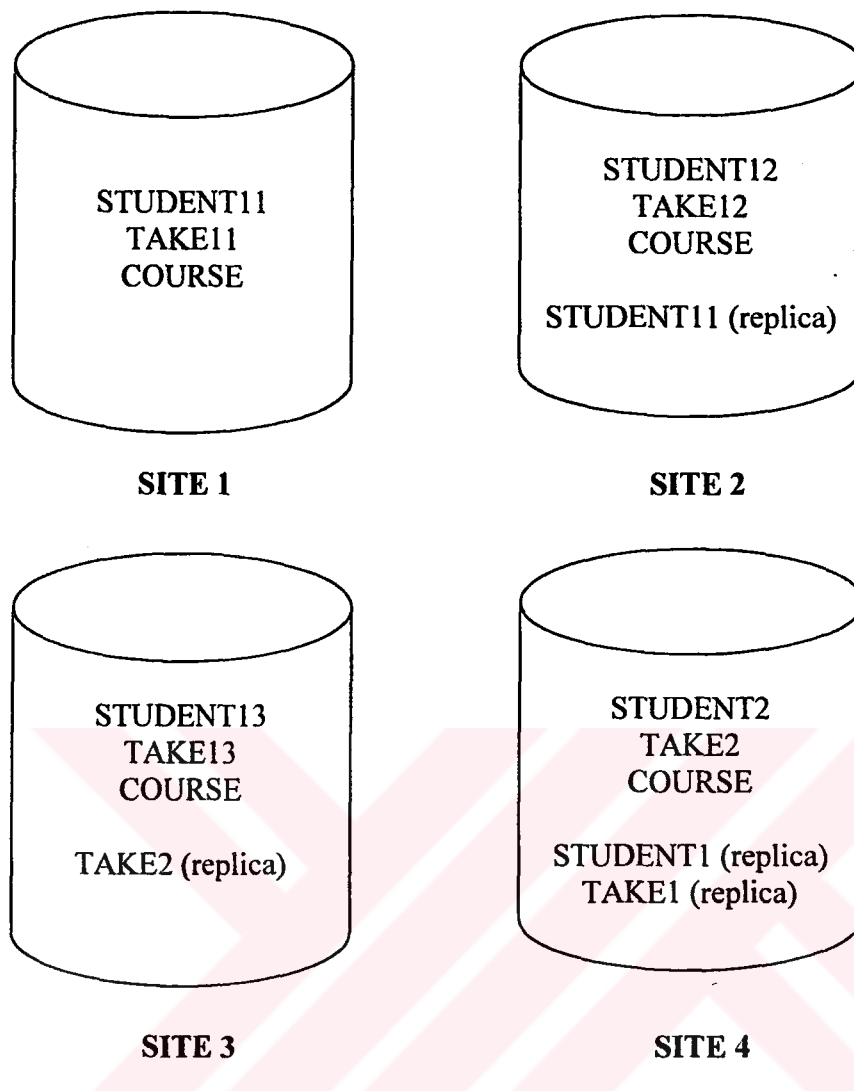


Figure 6.4 Allocation Schema of an example Student Database

6.2.1. Data Retrieval Queries

6.2.1.1. Selection from one table

When a query is produced and the Execute button is clicked, the Fragmenter procedure is called. The main role of the Fragmenter is converting a user query which is unaware of being in a distributed database environment, thus does not contain any fragmentation information; into a distributed query by getting help from the Name Server (Figure 6.5).

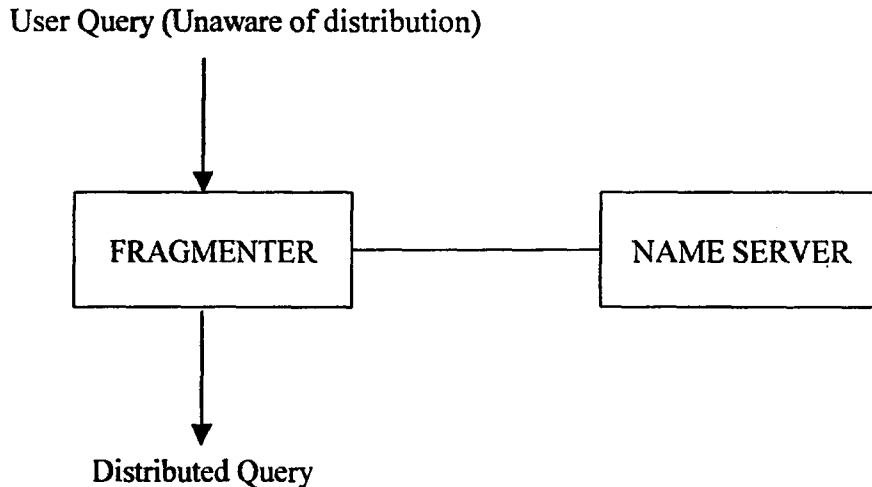


Figure 6.5 The main role of Fragmenter

If the query has no condition part, then the Fragmenter directly calls the `form_fragment_tree` procedure. This is a recursive procedure that finds all fragments of a table in different depths. By forming a fragmentation tree, it finds where these fragments physically exist and by taking the alias names, it forms the distributed query. If the query has a condition part (this is a part of the where condition except the join statement. I called this part as a `value_condition`), it is not meaningful to search for all fragments of a selected table. Because, while some fragments meet the given condition, the others do not. At this point, the Fragmenter first calls the `parse_val_cond` procedure. This procedure follows these steps:

- Takes the `value_condition` of the query which is in the form of :
predicate1 or/and predicate2 or/and predicate3 or/and
- Parses this value condition into two arrays: predicates and operators. After this parsing, we have an array called Predicates, containing predicate1,2,3, and an array called Operators, containing or/and logical operators in the order of entrance.

Example 1:

If the query **select * from student where student.student_id>=12 and student.student_id<25 and student.dept="Math"** is issued, `parse_val_cond` procedure parses the value condition of a query into predicates and operators as can be seen below:

`Predicates[1]= 'Student.student_id>=12'`

`Predicates[2]= 'Student.student_id<25'`

`Predicates[3]= 'Student.dept="Math"'`

`Operators[1]= 'and'`

`Operators[2]= 'and'`

Then fragmenter calls the Compare procedure for each predicate in the array `Predicates`. The aim of the Compare procedure is to find fragments of a table that meets predicates or conditions. Compare procedure executes as follows :

The first predicate `Student.student_id>=12` is compared with the fragmentation condition of `Student1` which is `dept=math`, these conditions are on different attributes, so we have to look to the subfragments of `Student1`. The fragmentation condition of `Student11` is on the same attribute, so we have to compare their values. After the comparison process, it is seen that `Student11` is not a suitable fragment. Then we do the same process with `student12`, this fragment is suitable. Then we are comparing the first predicate with `Student13` and this is also suitable. Then we are comparing the first predicate with `Student2`. They are not on the same attribute and this fragment has no other sub fragments. Although it's fragmentation attribute is different from predicate, this is also a suitable fragment, at least we can not say that there will not be any tuple that meets our condition in this fragment. Since the fragmentation condition and the given predicate condition are not on the same attribute, in this fragment there might be some records meeting our condition. The

same steps are repeated for all predicates. At the end, all suitable fragments are recorded into the `save_array` that can be seen below:

```
save_array[1]:=[Student.student_id>=12,Student12]
save_array[2]:=[Student.student_id>=12,Student13]
save_array[3]:=[Student.student_id>=12,Student2]
save_array[4]:=[Student.student_id<25,Student11]
save_array[5]:=[Student.student_id<25,Student12]
save_array[6]:=[Student.student_id<25,Student13]
save_array[7]:=[Student.student_id<25,Student2]
save_array[8]:=[Student.dept="math",Student11]
save_array[9]:=[Student.dept="math",Student12]
save_array[10]:=[Student.dept="math",Student13]
```

Now, it is the time to use logical operators, so the fragmenter calls `find_op_priorities` procedure. This procedure has the aim of binding predicates with logical operators. The “AND” logical operator has a priority on the “OR” logical operator. So at the end of this procedure, it is obvious that, `predicate1` and `predicate2` will be concatenated with the “AND” operator, then the result and `predicate3` will be concatenated again with the “AND” operator. These steps are recorded in array `Arr`, the end result will be `A2`:

```
Arr[1]:=[Student.st_id>=12, Student.st_id<25, and, A1]
Arr[2]:=[Student.dept="Math", A1, and, A2]
```

The information kept in `Arr` can be explained as follows:

- N-ary logical operations are reduced to binary logical operations.
- If two predicates are concatenated using the “AND” operator, this means that fragments that meet both `predicate1` and `predicate2` can be found by intersecting the set of fragments that meet `predicate1` and the set of fragments

that meet predicate2. The fragments found in the result of the intersection are kept in an array called Result_array.

- If two predicates are concatenated using the “OR” operator, this means that fragments that meet the condition “predicate1 OR predicate2” can be found by taking the union of the set of fragments that meet predicate1 and the set of fragments that meet predicate2. The fragments found in the result of union is kept in an array called Result_array.
- An intermediate result and a predicate can be concatenated using the “AND” operator, meaning that, it is necessary to intersect the set of fragments in the intermediate result and the set of fragments meeting the predicate.
- An intermediate result and a predicate can be concatenated using the “OR” operator, meaning that, it is necessary to take the union of the set of fragments in the intermediate result and the set of fragments meeting the predicate.
- There can be at most two intermediate result sets in this system since we are reducing the operations to binary operations by taking the priorities into account, and these intermediate results can be concatenated using the “OR” operator, which means that it is necessary to take the union of two intermediate fragment sets.
- Two intermediate results can not be concatenated using the “AND” operator, because, in our interface, it is not possible to create a condition like this: (p1 or p2) and (p3 or p4), since we do not support parentheses. As a result, in our system it is impossible to see the situation below in an Arr array :

Arr[1]:=[predicate1,predicate2,or,A1]

Arr[2]:=[predicate3,predicate4,or,A2]

Arr[3]:=[A1,A2,and,A3]

By using our interface, if someone gives this condition in the same order above, without parentheses, P1 or p2 and p3 or p4, the Find_op_priorities procedure will form the Arr array like this:


```

Arr[1]:=[p2,p3,and,A1]
Arr[2]:=[p1,A1,or,A2]
Arr[3]:=[A2,p4,or,A3]

```

When we return to our first example, Predicate[1] and Predicate[2] will be concatenated using the “AND” operator. So the set of fragments whose conditions are meeting predicate1, and the set of fragments meeting predicate2 will be intersected. So the distributed query will be executed on these intersecting fragments which are student12, student13, student2. These fragments are obtained for the A1 intermediate result. Then these fragments which we found for A1 will be again intersected with the fragments of predicate[3]. Resulting fragments are kept in result_array.

After these steps, the Fragmenter calls the find_fragments procedure. This procedure’s job is to find intersecting fragments for the “AND” operator and unions of the fragments for the “OR” operator. The find_fragments procedure gets the help of these procedures:

- **is_predicate**: Finds if the operands of AND/OR operators are a predicate or an intermediate result.
- **intersect_predicates**: Finds intersecting fragments of two predicates.
- **intersect_pre_result**: Finds intersecting fragments of an intermediate result and a predicate.
- **union_predicates**: Finds the union of fragments of two predicates.
- **union_pre_result**: Finds the union of fragments of intermediate result and the predicate.
- **union_results**: Finds the union of two intermediate results, i.e. result_array[1], result_array[2].

The last step is evaluating these fragments existing in result_array and forming the distributed query by finding each of these fragment aliases. In Figure 6.6 the screen of an intelligent interface, which is used to produce selection queries can be seen.

Selection

Choose table(s)

student
take
course

→ Add
← Remove

student

Quit

Choose attribute(s)

student.stid
student.sname
student.address

→ Add
← Remove

student.stid
student.sname

Where Condition

Join

AND CANCEL

Condition

student.stid < 12

AND OR CANCEL

student.dept="math" and student.stid>5 and student.stid<12

SQL Form

SQL string

select student.stid, student.sname from student where student.dept="math" and student.stid>5 and student.stid<12

Distributed Execution SQL string

select stid, sname from ":site1:STUDENT11" where dept="math" and stid>5 and stid<12 union select stid, sname from ":site2:STUDENT12" where dept="math" and stid>5 and stid<12

Execute Query

Output

stid	sname
10	melahat celik
11	meliha baskömürçü

User Query (Unaware of distribution)

Distributed Query (Converted by Fragmenter)

Figure 6.6 An Interface for Data Retrieval Queries

As can be seen from the example query in Figure 6.6, Fragmenter converts the user query into a distributed query and finds suitable fragments; accordingly the best suitable sites to execute the query. In data retrieval queries, when it is possible to retrieve data from multiple replicas located at different sites, choosing the optimal site is an optimization issue including load balancing and other networking concepts which is a thesis topic individually. Because of this wide range, in this thesis, the site choosing process is made randomly, no other optimization is considered. In Figure 6.7 a user query involving the Course table can be seen. Since the Course table is replicated at four sites, Fragmenter has to make a decision on the site to execute the query.

Selection

Choose table(s)

student	→ Add	course
take	← Remove	
course		

Choose attribute(s)

course.cid	→ Add	course.cid
course.cname	← Remove	course.cname

Where Condition

Join

	↔		AND	CANCEL
--	---	--	-----	--------

Condition

			AND	OR	CANCEL
--	--	--	-----	----	--------

SQL string

select course.cid,course.cname from course

Distributed Execution SQL string

select course.cid,course.cname from ":site1:COURSE"

Execute Query

Output

cid	cname
3	theory of computer
4	DBMS
5	software engineering
6	programming language

Site 1 was chosen to execute the query

Figure 6.7 Selection from a replicated table

6.2.1.2. Selection from multiple tables -Join Operation-

In Chapter 3 section 3.6, four layers of query processing were explained. One of them was the localization layer. This layer translates a query on global relations into a query expressed on physical fragments. In this layer, a global relation is first reconstructed by applying the reconstruction (or reverse fragmentation) rules and deriving a relational algebra program whose operands are the fragments. This process is called a localization program. The query obtained this way is called a generic query.

Example 2:

By using the fragmentation schema of Student table, a generic query for **Select * from Student where dept="Ceng"** can be seen in figure 6.8:

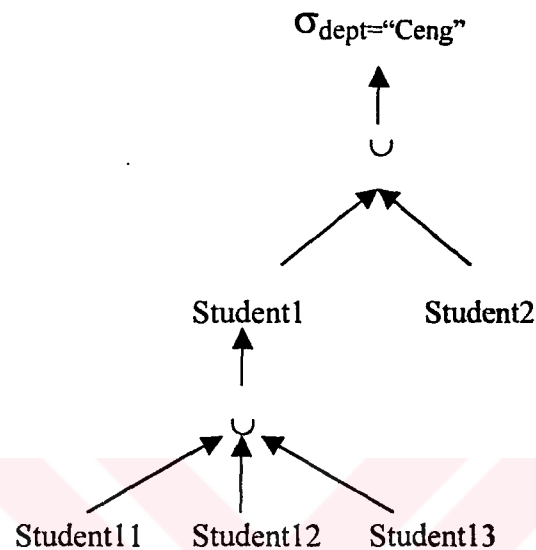


Figure 6.8 Generic Query

As can be seen in Figure 6.8, and according to the explanations made previously, to get the correct result of this query, it is not necessary to look at Student1, or neither of its fragments, since the fragmentation condition and value condition given in the query are contradicting. In general, generating a generic query is inefficient, because important restructurings and simplifications of the generic query can still be made. To generate simpler and optimized queries, reduction techniques must be applied to the generic query. A reduced query can be seen in Figure 6.9 for this example:

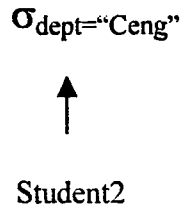


Figure 6.9 Reduced Query

In queries where multiple tables are involved, the join operation comes on the scene. Join is the most expensive operation even in centralized databases, thus in a distributed database environment, this operation gains more importance and needs optimization. This optimization process is affected by too many parameters in a distributed database environment and has a considerable wide range. In this thesis, when a selection query without a join operation is issued, the reduced query is produced. But because of the complexity of the join operation, it is not optimized and made by constructing the generic query of the tables involved, no reduction or optimization is made on a generic query. To process join queries, an Intelligent Interface reconstructs the global relations from their fragments by union operation, then executes the queries on these global relations. Since this method requires a lot of data movement, and join operation has to be applied on very large data sets, this is not a cost effective method especially in very large distributed databases. Further studies can concentrate on join operation and optimizations can be made. A user query including a join operation can be seen in Figure 6.10.

Selection

Choose table(s)

student	→ Add ← Remove	student
take		take
course		course

Choose attribute(s)

take.score	→ Add ← Remove	student.sname
course.cid		course.cname
course.cname		take.score

Where Condition

Join

take.cid	<==>	course.cid	AND	CANCEL
student.stid=take.stid and take.cid=course.cid				

Condition

			AND	OR	CANCEL
--	--	--	-----	----	--------

SQL string

```
select student.sname,course.cname,take.score from student,take,course where
student.stid=take.stid and take.cid=course.cid
```

Distributed Execution SQL string

Execute Query

Output

sname	cname	score
ilknur şanlı	theory of computer	90
ilknur şanlı	data structures	55
ali başkömürcü	data structures	65
ali başkömürcü	DBMS	95

Figure 6.10 Query with a Join Operation

6.2.2. Data Manipulation (Insert, Update, Delete) Queries

In this thesis, any selection statement can be executed in any type of the distributed database, designed by using primary and derived horizontal fragmentation and replication, as long as the distribution information is recorded in the Name Server. But, data manipulation operations are more application dependent and although it is not impossible, it is difficult to design a general interface for these operations. As a result, for data manipulation operations, I studied on an example Student database application, therefore an interface prepared for insert, update, delete operations is dependent on my application.

By using the data manipulation interface prepared for an example Student database application, it is possible to insert, update or delete a tuple. While making these operations, just like in selection queries, query reduction techniques are used, meaning that changes are propagated only at involving fragments. Another important issue here is, considering replications in the system. It is not sufficient only to find an involving fragment to execute any query, but it is also necessary to propagate the changes at all sites, which contain replicas of that fragment. For this purpose a queue mechanism is used. When a site involved in the query is unreachable for that moment, the update, insert or delete is recorded in the queue for that site. A timer of the queue continuously runs and checks the sites in the queue if they become available. If it finds any site being available, it executes the process in the queue for that site, and deletes that queue record immediately. Of course this deferred propagation of updates may result in temporarily inconsistent data at replicated sites, but in our example application, real time, exact consistency is not so important. Copies therefore can be temporarily inconsistent, but over time the data should converge to the same values at all sites.

In Figure 6.11, an insertion into Student table can be seen. The student_id of a tuple to be inserted is 6 and the department is “math”, according to the fragmentation schema, this tuple should be inserted into the fragment Student11. This fragment physically exists at Site 1 and it is replicated at Site 2. At the same time, the fragment Student1 which includes Student11 is at Site 4. Therefore, this insertion operation should be propagated at Site 1, Site 2 and Site 4.

Insert & Update & Delete Student Entry

Student id :

Name :

Address :

Department :

New Record

Insert

Update

Delete

Refresh

Stid	Sname	Address	Dept
1	ilknur şanslı	karşıyaka	math
2	ali başkömürcü	güzelyalı	math
3	özlem şanslı	hatay	math
4	saadet şanslı	karşıyaka	math
5	hüseyin şanslı	karşıyaka	math
10	melahat çelik	güzelyalı	math
11	meliha başkömürcü	güzelyalı	math
12	yılmaz çelik	güzelyalı	math
13	özge çelik	güzelyalı	math
20	vedat ışık	gaziemir	math

Figure 6.11 Insertion into the Student table

After an insert operation at Figure 6.11, the user of the Intelligent Interface will see the result, which is shown in Figure 6.12.

Insert & Update & Delete Student Entry

Student id:

Name:

Address:

Department:

New Record

Insert

Update

Delete

Refresh

Stid	Sname	Address	Dept
1	ilknur şanslı	karşıyaka	math
2	ali başkömürücü	güzelyalı	math
3	özlem şanslı	hatay	math
4	saadet şanslı	karşıyaka	math
5	hüseyin şanslı	karşıyaka	math
6	şebnem oldaç	karşıyaka	math
10	melahat çelik	güzelyalı	math
11	meliha başkömürücü	güzelyalı	math
12	yılmaz çelik	güzelyalı	math
13	özge çelik	güzelyalı	math

Figure 6.12 The result of the insert operation from the user's point of view

As can be seen from Figure 6.12, an intelligent interface made update propagation transparent to the users. Although the users see the Student table as a whole in a grid, they are not aware of the fact that, these records came from different fragments of the Student table, which are distributed to different sites. The users also don't know the fact that an insert operation is propagated at three sites. The real results of this insert operation will be shown by using Database Explorer tool of Delphi. In Figure 6.13, inserted record can be seen at relevant fragments, which physically exist at Site 1, Site 2 and Site 4.

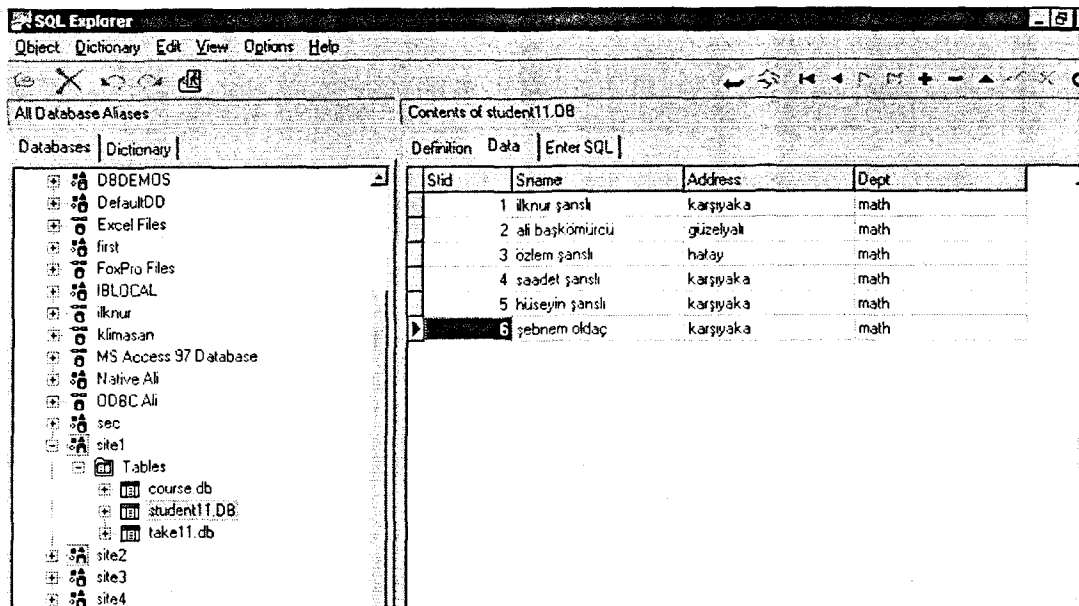


Figure 6.13.a The Student11 Fragment at Site 1

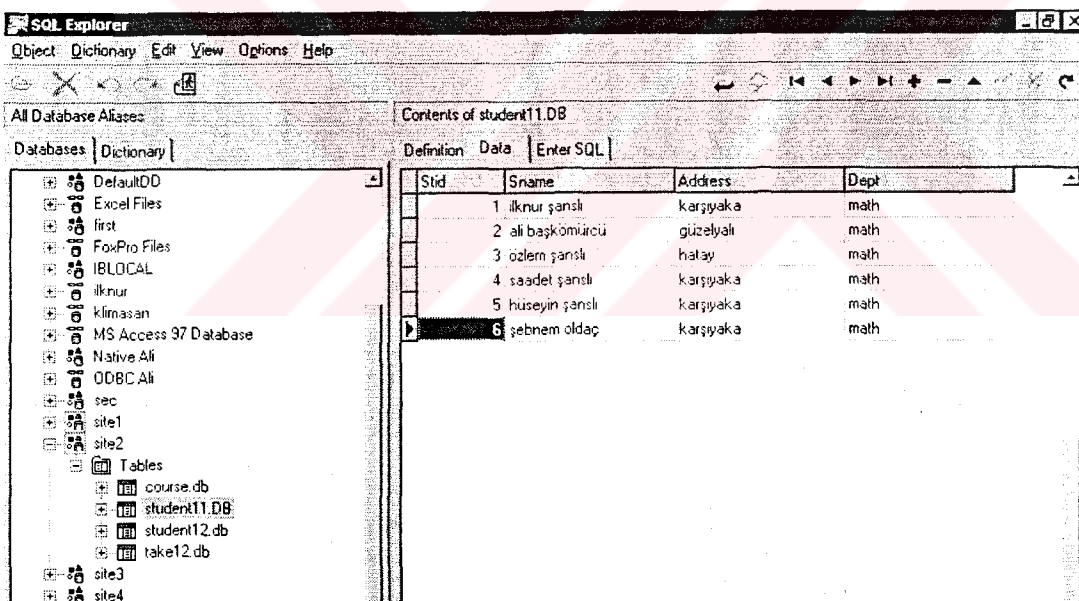
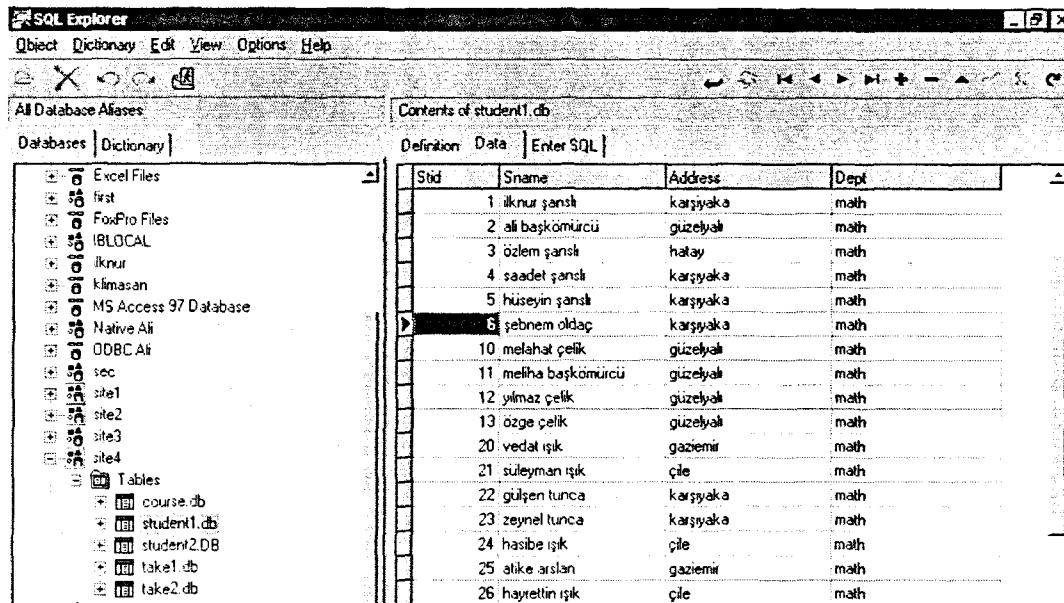


Figure 6.13.b The Student11 Fragment at Site 2



SQL Explorer

Object Dictionary Edit View Options Help

All Database Aliases

Databases Dictionary

Contents of student1.db

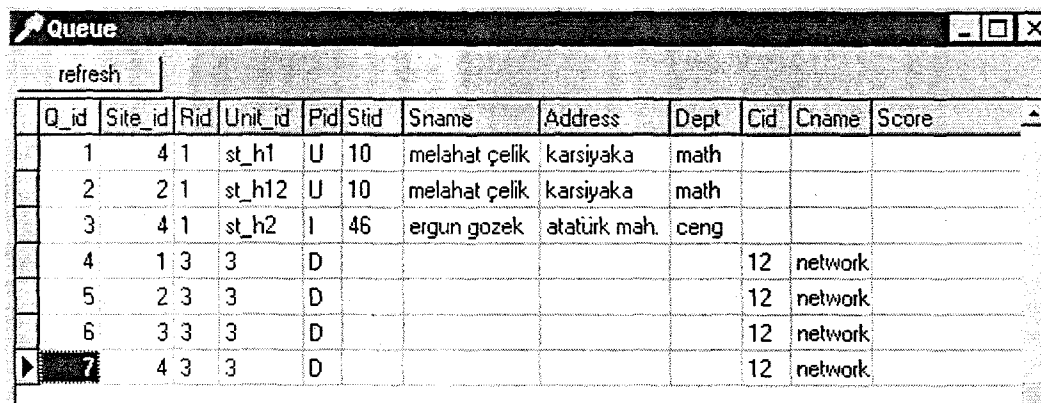
Definition Data Enter SQL

Stid	Sname	Address	Dept
1	ilknur şanlı	karsiyaka	math
2	ali başkomurcu	güzelyalı	math
3	özlem şanlı	halay	math
4	saadet şanlı	karsiyaka	math
5	hüseyin şanlı	karsiyaka	math
6	şebnem oldaç	karsiyaka	math
10	melahat çelik	güzelyalı	math
11	melih başkomurcu	güzelyalı	math
12	yılmaz çelik	güzelyalı	math
13	özge çelik	güzelyalı	math
20	vedat ışık	gaziemir	math
21	süleyman ışık	çile	math
22	gülşen tunca	karsiyaka	math
23	zeynel tunca	karsiyaka	math
24	hasibe ışık	çile	math
25	alıke arslan	gaziemir	math
26	hayrettin ışık	çile	math

Figure 6.13.c The Student1 Fragment at Site 4

Figure 6.13 Update Propagation Example

If any site involved in an insert, update or delete operation is unreachable for that moment, this operation will be kept in a queue for that site. In Figure 6.14 the records in the queue can be seen. Pid is the process id of an operation to be performed (I-insert, U-update, D-delete). Site_id and Unit_id shows the site_id and the unit_id on which an operation to be performed.



Queue

refresh

Q_id	Site_id	Rid	Unit_id	Pid	Stid	Sname	Address	Dept	Cid	Cname	Score
1	4	1	st_h1	U	10	melahat çelik	karsiyaka	math			
2	2	1	st_h12	U	10	melahat çelik	karsiyaka	math			
3	4	1	st_h2	I	46	ergun gozek	ataturk mah.	ceng			
4	1	3	3	D					12	network	
5	2	3	3	D					12	network	
6	3	3	3	D					12	network	
7	4	3	3	D					12	network	

Figure 6.14 Queue Mechanism

6.2.3. Pseudo Codes

Fragmenter()

If (has_no_join_part) and (has_no_condition_part) {selection from one table}

If table has fragments {Find all the fragments from Fragment table whose
parent is equal to the table}

For all fragments of the table

Form_Fragment_Tree { Divide the query into subqueries which will be
executed on different fragments}

Execute all the subqueries

If table has no fragments

Find where the table exists {Look at the Unit_Dist table}

If the table has replicas at more than one site

then find optimal site,

else find the site where the table exists

Execute the query

If (has_no_join_part) and has_condition_part

If table has fragments

Parse_Val_Cond { Divide the value condition into Predicates and
Operators arrays}

For all Predicates

For all Fragments

Compare { Output is Save_Array which has suitable fragments
for each predicate}

Find_Op_Priorities { Find the priorities of AND/OR operators, and
consequently find how the predicates will be
connected by logical operators – Output is Arr

Array}

Find_Fragments {Find the result Fragment Set to execute the query}

Execute the query on the fragments in Result_Array

If the table has no fragments

Find where the relation exists

Execute the query at that site

If has_join_part

For all tables involved in the query

Form_Fragment_Tree { Form the generic query for a table to generate
global relation}

Execute the query and get all the results of a global relation

Create a temporary table and fill it with the global data obtained

Execute the user query with join operation on these temporary tables which
represent global relations

Form_Fragment_Tree (Unit_id)

{A unit can be a relation or a fragment}

Search for a unit in Unit_Dist table

If found {It is a physical unit}

If it is replicated at more than one site

then choose the optimal site and take the alias_name

else take the alias_name of that site

Form the sub query

Add the sub query to the distributed query

If not found {It is a logical unit}

Find all the sub fragments of that unit

For all sub fragments

Form_Fragment_Tree(Sub_fragment_id)

Compare (Fr_id, Predicate)

Look to the derived attribute of the Fragment table if that fragment is derived from another fragment

If it is derived

 then att=fragmentation attribute of an owner relation

 else att=fragmentation attribute of a fragment itself

patt=attribute of the Predicate (selection formula)

If patt=att

 then make a value comparison to see whether that fragment is suitable to execute query or not

 If it is suitable then find_leaves (fr_id, predicate)

else

 If the fragment has sub fragments then

 For all sub fragments Compare (sub_fragment_id,predicate)

 If the fragment has no sub fragments then

 Record into Save_Array that the fragment is suitable for a predicate

Find_Leaves (fr_id, predicate)

Find all sub fragments of the fragment

If it has sub fragments

 then

 For all sub fragments

 Find_leaves (sub_fragment_id, predicate)

else

 Record into Save_Array that the fragment is suitable for a predicate

Find_Op_Priorities (Predicates_Array, Operators_Array)

According to the priority of AND operator to OR operator

Examine the Predicates Array and Operators Array

Find how the predicates are connected with logical operators

{Output is Arr array in the form of Arr [p1,p2, op, A_i]:

Arr [predicate1, predicate2, logical_operator, intermediate_result] or

Arr [intermediate_result, predicate, logical_operator, intermediate_result] or

Arr [predicate, intermediate_result, logical_operator, intermediate_result] or

Arr [intermediate_result, intermediate_result, logical_operator (OR),
intermediate_result]}

Find_Fragments (Arr_Array)

For all rows in the Arr array

If logical_operator is AND then

If p1=predicate, p2=predicate then intersect_predicates(p1,p2)

If p1=predicate, p2=intermediate_result then intersect_pre_result(p1)

If p1=intermediate_result, p2=predicate then intersect_pre_result(p2)

If logical_operator is OR then

If p1=predicate, p2=predicate then union_predicates(p1,p2)

If p1=predicate, p2=intermediate_result then union_pre_result(p1)

If p1=intermediate_result, p2=predicate then union_pre_result(p2)

If p1=intermediate_result, p2=intermediate_result then union_results()

{Intermediate result is kept in Result_Array, at the end the fragments found in the Result_Array will form the distributed query}

Intersect_Predicates(p1,p2)

Find the fragment set for the predicate p1 from Save_Array \rightarrow FS1

Find the fragment set for the predicate p2 from Save_Array \rightarrow FS2

Find $FS1 \cap FS2$ and record this set into Result_Array

Intersect_Pre_Result(p)

Find the fragment set for the predicate p from Save_Array \rightarrow FS1

RS= the set recorded in the Result_Array

Find $FS1 \cap RS$ and record this set into Result_Array

Union_Predicates(p1,p2)

Find the fragment set for the predicate p1 from Save_Array \rightarrow FS1

Find the fragment set for the predicate p2 from Save_Array \rightarrow FS2

Find $FS1 \cup FS2$ and record this set into Result_Array

Union_Pre_Result(p)

Find the fragment set for the predicate p from Save_Array \rightarrow FS1

RS= the set recorded in the Result_Array

Find $FS1 \cup RS$ and record this set into Result_Array

Union_Results()

RS1=the set recorded in the Result_Array[1]

RS2=the set recorded in the Result_Array[2]

Find $RS1 \cup RS2$ and record this set into Result_Array

{The number of columns in the Result_Array depends on the number of elements found as a result of set union or intersection operations. But there can be at most to rows in the Result Array. We can see this fact in an example. Assume that the user query has the condition part like:

p1 and p2 or p3 and p4 and p5 or p6

Arr[1]=[p1,p2,and,A1]→ The result A1 will be at Result_Array[1]

Arr[2]=[p3,p4,and,A2]→ The result A2 will be at Result_Array[2]

Arr[3]=[A2,p5,and,A3]→ The set in the Result_Array[2] and the set of fragments which are suitable with p5 will be intersected. The result will be kept in Result_Array[2]

Arr[4]=[A1,A3,or,A4]→ The set in the Result_Array[1] union
The set in the Result_Array[2] will be kept in the Result_Array[1]. Result_Array[2] will be cleared

Arr[5]=[A4,p6,or,A5]→ The union of the set in the Result_Array[1] and the set of fragments which are suitable with P6 will be taken. The result will be kept in Result_Array[1]}

Process_Queue()

Repeat

For each record in the queue

 If the site becomes available then

 If process_id='I' then Insert_Entry()

 If process_id='U' then Update_Entry()

 If process_id='D' then Delete_Entry()

 Delete the record from the queue



CONCLUSIONS

In this thesis, I tried to implement an intelligent interface, which is a prototype of a Distributed Database Management System. As a result of the time limitation of this study, the complexity and the wide range it includes, I made some restrictions on the functions of this system. I believe that this study will give a general framework to understand the topic “Distributed Database Management System”, and the prototype I implemented will also help to imagine functions of such a system. Another important issue for this study is, aiming to be the basis for future works on this topic.

This study can be extended by future works on the topics below:

- When there is more than one site where the query can be executed, the site choosing process can be optimized by considering the network load of the system.
- Reduced query plans can be generated instead of the generic query, for the join operation.
- The user interface can be generalized, so that it can run on all types of distributed database systems for data manipulation operations. (An application independent interface can be implemented.)
- A module for the configuration of the Name Server can be implemented. By using this module designers of a distributed database system can enter data which introduce the distributed database system to an intelligent interface. In the interface introduced in this thesis, this process is made directly by using the Name Server database, i.e, there is no interface to manipulate data in the Name Server database.

- A module for the design of a distributed database can be implemented. The interface introduced in this thesis is running on a distributed database designed according to some restrictions, it can not design a distributed database. By using this design module, it can be possible to fragment a table according to a given criteria, to reconstruct a table from its fragments, to replicate a fragment or a table at sites, or to allocate fragments or tables to sites.
- In the design of a distributed database, the Name Server can be replicated at all sites, or it can be located at only one site. Since the Name Server database is the brain of an intelligent interface, this selection is a very important design issue. When replicating the Name Server at more than one site, there should be mechanisms to synchronize data at all Name Servers. On the other hand, when the Name Server is located at only one site, since all transactions need the data on the Name Server, there might be a bottleneck in the system. It is strongly recommended that, the brain of a distributed database should not be a central, single point of failure, this would be a dilemma. Therefore it is more suitable to distribute the Name Server database and implement strict mechanisms to keep all Name Servers consistent. Here, for data changes on Name Servers, all or nothing approach can be used.
- The same design alternatives for the Name Server database is also valid for a queue mechanism. Each site can maintain its own local queue, or there can be only one queue mechanism at only one site. Each of these design alternatives brings some advantages and disadvantages, just like in Name Server location selection case.

The topics discussed above are future works that can be based on my research. Beside these, it is also important to consider the current and new trends in distributed database technology, which are parallel servers, distributed knowledge bases and distributed object-oriented databases. It will be a great honour for me, that my study would be the basis for further studies on these topics.

REFERENCES

[Casavant & Singhal, 1994]. Thomas L. Casavant & Mukesh Singhal. Readings in Distributed Computing Systems. IEEE Computer Society Press.

[Chen & Roussopoulos, 1994]. Chungmin Melvin Chen & Nick Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994. ACM Press 1994, SIGMOD Record 23(2), June 1994.

[COT5200]. Distributed Database Systems (Advanced topics in DS) lecture notes. Monash University, Faculty of Information Technology, School of Computer Science & Software Engineering. Internet address:
<http://www.ct.mpnash.edu.au/~Zaslav/cot5200-link/>

[Delis & Roussopoulos, 1992]. Alexios Delis & Nick Roussopoulos. Performance and Scalability of Client-Server Database Architectures. 18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings.

[Delis & Roussopoulos, 1994]. Alex Delis & Nick Roussopoulos. Management of Updates in the Enhanced Client-Server DBMS. Proceedings of the 14th International Conference on Distributed Computing Systems. June 21-24, 1994, Poznan, Poland, IEEE Computer Society Press.

[IBM_DRDA]. DRDA (Distributed Relational Database Architecture). IBM Software. Internet address: <http://www.software.ibm.com/data/drda.html>.

[Ibrahim et al., 1998]. H. Ibrahim, W.A. Gray and N.J. Fiddian. Optimizing Fragment Constraints. Proceedings of the 9th International Workshop on Database and Expert Systems Applications, August 24-28, 1998, Vienna, Austria. IEEE Computer Society Press.

[March & Rho, 1995]. Salvatore T. March & Sangkyu Rho. Allocating Data and Operations to Nodes in Distributed Database Design. IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 2, April 1995.

[MARIPOSA_manual]. Mariposa Distributed Database Management System User Manual v.1.0. Internet address: <http://mariposa.cs.berkeley.edu/download.html>.

[Muthuraj et al., 1993]. Jaykumar Muthuraj, Sharma Chakravarthy, Ravi Varadarajan, Shamkant B. Navathe. A Formal Approach to the Vertical Partitioning Problem in Distributed Database Design. Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems, San Diego, CA, USA, January 20-23, 1993. IEEE Computer Society Press.

[Oracle 7]. Oracle 7, Release 7.3.4, Documentation Library.

[Ozkarahan, 1997]. Esen Ozkarahan. Database Mangement Concepts, Design and Practice (2nd ed.). Saray Medical Publication.

[Ozsu & Valduriez, 1991]. M. Tamer Ozsu & Patrick Valduriez. Principles of Distributed Database Systems. Prentice-Hall, Inc.

[Ozsu & Valduriez_notes, 1999]. M. Tamer Ozsu & Patrick Valduriez. Principles of Distributed Database Systems, (2nd ed.) Viewable Notes on Distributed Transaction Management.

Internet address: <http://web.cs.ualberta.ca/~database/ddbook/notes/Transaction/index.htm>.

[Reddy & Kitsuregawa, 1998]. P. Krishna Reddy & Masaru Kitsuregawa. Reducing the Blocking in Two-Phase Commit Protocol Employing Backup Sites. Proceedings of the 3rd International Conference on Cooperative Information Systems, New York, USA, August 20-22, 1998. IEEE Computer Society Press.

[Rennhackkamp, 1998]. Martin Rennhackkamp. Peerdirect-Heterogeneous Replication comes of age. DBMS Magazine, May 1998. DBMS Online.

[Richter, 1994]. Jane Richter. Distributing Data. Byte Special Report, June 1994.

[Shirota et al., 1999]. Yukari Shirota, Atsushi Uzawa, Hiroko Mano, Takashi Yano. The ECHO Method : Concurrency Control Method for a Large-Scale Distributed Database. Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Australia. IEEE Computer Society Press.

[Sidell et al., 1996]. Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker and Andrew Yu. Data Replication in Mariposa. Proceedings of the 12th International Conference on Data Engineering, Feb. 26-March 1, 1996, New Orleans, Louisiana, USA. IEEE Computer Society Press.

[Sybase1]. Alex Moissis. Sybase Replication Server: A Practical Architecture for Distributing and Sharing Corporate Information. Sybase Inc. Internet address: http://www.sybase.com/products/datamove/repserver_wpaper.html.

[Sybase2]. Sybase Replication Server. Sybase Inc. Internet address: <http://www.sybase.com/products/system11/repserver.html>.

[Tanenbaum, 1995]. Andrew S. Tanenbaum. Distributed Operating Systems. 1995 by Prentice-Hall, Inc.

[Ullman, 1982]. Jeffrey D. Ullman. Principles of Database Systems. (2nd ed.). Computer Science Press, Inc.

[Zhang et al., 1999]. Zhili Zhang, William Perrizo, Victor T.-S. Shi. Atomic Commitment in Database Systems over Active Networks. Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Australia. IEEE Computer Society Press.

