NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

DISTRIBUTED DATABASE SYSTEMS

Graduation Project
COM – 400

Student : Fatoş Yuvarlak (970720)

Supervisor: Assoc.Prof.Dr.Rahib ABIYEV

Nicosia-2002

ACKNOWLEDGEMENT

First of all I would like to my Graduation Project Supervisor Assist.Prof.Dr.Rahib Abiyev who is a patient & very appreciating personality.He has guided me with a keen interest and helped me by all means.Dr.Rahib,thanks for your continual support.

I would like thank all my teachers in the Near East University, including Faculty of Engineering Dean Prof.Dr.Fahreddin Mamedov, Deparment of Computer Engineering Chairmen Assist.Prof.Dr.Adnan Khasman, Student Advisors Miss. Besime Erin and Mr. Tayseer Alshanableh and all the Staff of the Faculty of Engineering, and special thanks to the Vice-President of Near East University Assoc.Prof.Dr. Şenol Bektaş for given me the opportunity to experience this remerkable Institute that have showed me preliminary steps towards my professional carrier.

And, I want to say thanks to my father İhsan YUVARLAK for providing both moral and financial support that made the completion of the project .

ABSTRACT

Distributed database system (DDBS) technology is the union of what appear to be two diametrically opposed approaches to data processing: database system and computer network technologies. Database systems have taken us from a paradigm of data processing, in which each application defined and maintained its own data to one in which the data is and administered centrally. This new orientation results in data independence, whereby the application programs are immune to changes in the logical or physical organization of the, and vice versa.

In the following chapters I will explain in detailed information about all of them DISTRIBUTED DATABASE SYSTEMS.

TABLE OF CONTENTS

INTRODUCTION

Distributed database system (DDBS) technology is one of the major recent developments in the database systems area. There are claims that in the next ten years centralized database managers will be an "antique curiosity" and most organizations will move toward distributed database managers. The intense interest in this subject in both the research community and the commercial marketplace certainly supports this claim. The extensive research activity in the last decade has generated results that now enable the introduction of commercial products into the market place.

Distributed database system (DDBS) technology is the union of what appear to be two diametrically opposed approaches to data processing: database system and computer network technologies. Database systems have taken us from a paradigm of data processing, in which each application denned and maintained its own data, to one in which the data is denned and administered centrally. This new orientation results in data independence, whereby the application programs are immune to changes in the logical or physical organization of the data, and vice versa.

One of the major motivations behind the use of database systems is the desire to integrate the operational data of an enterprise and to provide centralized, thus controlled access to that data. The technology of computer networks, on the other hand, promotes a mode of work that goes against all centralization efforts. At first glance it might be difficult to understand how these two contrasting approaches can possibly be synthesized to produce a technology that is more powerful and more promising than either one alone. The key to this understanding is the realization that the most important objective of the database technology is integration, not centralization. It is important to realize that either one of these terms does not necessarily imply the other. It is possible to achieve integration without centralization, and that is exactly what the distributed database technology attempts to achieve.

CHAPTER ONE

DISTRIBUTED DATA PROCESSING

The term distributed processing (or distributed computing) has been used to refer to such diverse systems as multiprocessor systems, distributed data processing, and computer networks. Here are some of the other terms that have been used synonymously with distributed processing: distributed function, distributed computers or computing, networks, multiprocessors / multi computers, satellite processing/satellite computers, backend processing, dedicated/special-purpose computers, time-shared systems, and functionally modular systems.

Some degree of distributed processing goes on in any computer system, even on single-processor computers. Starting with the second-generation computers, the central processing unit (CPU) and input/output (I/O) functions have been separated and overlapped. This separation and overlap can be considered as one form of distributed processing. However, it should be quite clear that what we would like to refer to as distributed processing, or distributed computing, has nothing to do with this form of distribution of functions in a single-processor computer system.

Distributed computing system states is a number of autonomous processing elements (not necessarily homogeneous) that are interconnected by a computer network and that cooperate in performing their assigned tasks. The "processing element" referred to in this definition is a computing device that can execute a program on its own.

One fundamental question that needs to be asked is: What is being distributed? One of the things that might be distributed is the processing logic. In fact, the definition of a distributed computing system given above implicitly assumes that the processing logic or processing elements are distributed. Another possible distribution is according to function. Various functions of a computer system could be delegated to various pieces of hardware or software. A third possible mode of distribution is according to data. Data used by a number of applications may be distributed to a number of processing sites. Finally, control can be distributed. The control of the execution of various tasks might be distributed instead of being performed by one computer system.

From the viewpoint of distributed database systems, these modes of distribution are all necessary and important. In the following sections we talk about these in more detail.

Distributed computing systems can be classified with respect to a number of criteria. Bochmann lists some of these criteria as follows: degree of coupling, interconnection structure, interdependence of components, and synchronization between components [Bochmann, 1983]. Degree of coupling refers to a measure that determines how closely the processing elements are connected together. This can be measured as the ratio of the amount of data exchanged to the amount of local processing performed in executing a task. If the communication is done over a computer network, there exists weak coupling among the processing elements. However, if components are shared, we talk about strong coupling. Shared components can be either primary memory or secondary storage devices. As for the interconnection structure, one can talk about those cases that have a point-to-point interconnection between processing elements, as opposed to those, which use a common interconnection channel. We discuss various interconnection structures. The processing elements might depend on each other quite strongly in the execution of a task, or this interdependence might be as minimal as passing messages at the beginning of execution and reporting results at the end. Synchronization between processing elements might be maintained by synchronous or by asynchronous means. Note that some of these criteria are not entirely independent. For example , if the synchronization between processing elements is synchronous, one would expect the processing elements to be strongly interdependent, and possibly to work in a strongly coupled fashion.

The distributed processing better corresponds to the organizational structure of today's widely distributed enterprises, and that such a system is more reliable and more responsive. Data can be entered and stored where it is generated, without any need for physical (manual) movement. Furthermore, building a distributed system might make economic sense since the costs of memory and processing elements are decreasing continuously.

The fundamental reason behind distributed processing is to be better able to solve the big and complicated problems, by using a variation of the well-known divide and-conquer rule. If the necessary software support for distributed processing can be developed, it might be possible to solve these complicated problems simply by dividing

them into smaller pieces and assigning them to different software groups, which work on different computers and produce a system that runs on multiple processing elements but can work efficiently toward the execution of a common task.

This approach has two fundamental advantages from an economics standpoint. First, we are fast approaching the limits of computation speed for a single processing element. The only available route to more computing power, therefore, is to employ multiple processing elements optimally. This requires research in distributed processing as denned earlier, as well as in parallel processing, which is outside the scope. The second economic reason is that by attacking these problems in smaller groups working more or less autonomously, it might be possible to discipline the cost of software development. Indeed, it is well known that the cost of software has been increasing in opposition to the cost trends of hardware.

Distributed database systems should also be viewed within this framework and treated as tools that could make distributed processing easier and more efficient. It is reasonable to draw an analogy between what distributed databases might offer to the data processing world and what the database technology has already provided. There is no doubt that the development of general-purpose, adaptable, efficient distributed database systems will aid greatly in the task of developing distributed software.

1.1 DISTRIBUTED DATABASE SYSTEM

We can define a distributed database as a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database management system (distributed DBMS) is then defined as the software system that permits the management of the DDBS and makes the distribution transparent to the users. The two important terms in these definitions are "logically interrelated" and "distributed over a computer network." They help eliminate certain cases that have sometimes been accepted to represent a DDBS.

First, a DDBS is not a "collection of files" that can be individually stored at each node of a computer network. To form a DDBS, files should not only be logically related, but there should be structure among the files, and access should be via a common interface. It has sometimes been assumed that the physical distribution of data

is not the most significant issue. The proponents of this view would therefore feel comfortable in labeling as a distributed database two (related) databases that reside in the same computer system. However, the physical distribution of data is very important. It creates problems that are not encountered when the databases reside in the same computer. Note that physical distribution does not necessarily imply that the computer systems be geographically far apart; they could actually be in the same room. It simply implies that the communication between them is done over a network instead of through shared memory, with the network as the only shared resource.

The definition above also rules out multiprocessor systems as DDBSs. A multiprocessor system is generally considered to be a system where two or more processors share some form of memory, either primary memory, in which case the multiprocessor is called tightly coupled, or secondary memory, when it is called loosely coupled. Sharing memory enables the processors to communicate without exchanging messages. With the improvements in microprocessor and VLSI technologies, other forms of multiprocessors have emerged with a number of microprocessors connected by a switch .

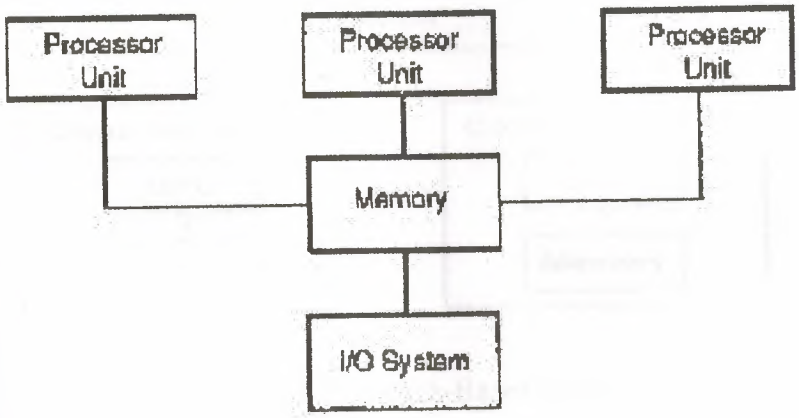

Figure 1.1: Tightly-Coupled Multiprocessor

Another distinction that is commonly made in this context is between shared everything and shared-nothing architectures. The former architectural model permits.

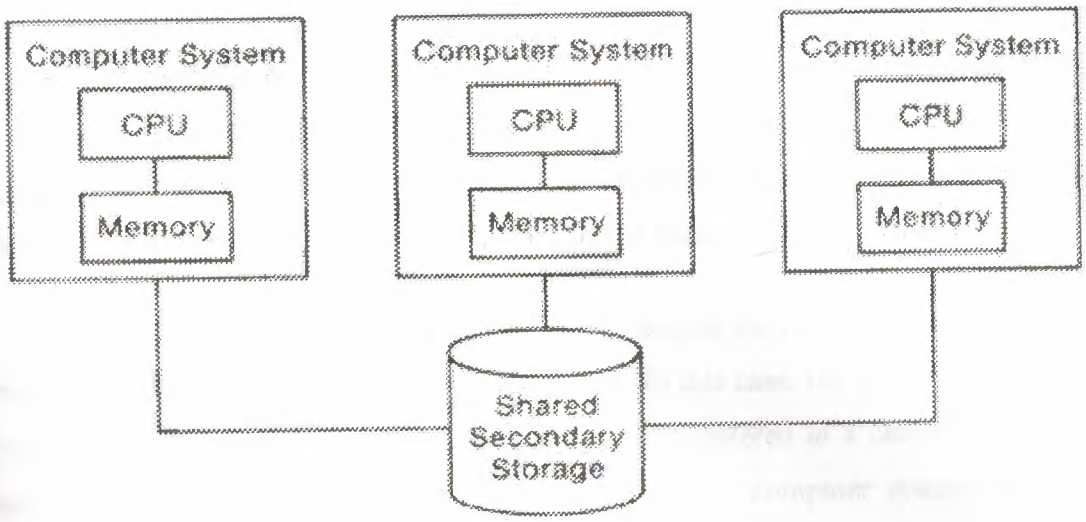

Figure 1.2: Loosely-Coupled Multiprocessor

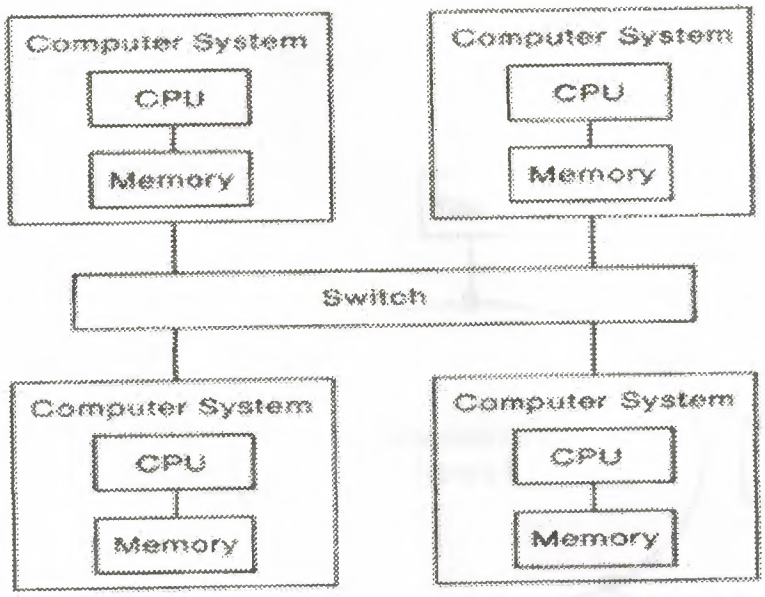

Figure 1.3: Switch-Based Multiprocessor System

each processor to access everything (primary and secondary memories, and peripherals) in the system and covers the three models that we described above. The shared nothing architecture is one where each processor has its own primary and secondary memories as well as peripherals, and communicates with other processors over a very high speed bus. In this sense the shared-nothing multiprocessors are quite similar to the distributed environment that we consider in this book. However, there are differences between the interactions in multiprocessor architectures and the rather loose interaction that is common in distributed computing environments. The fundamental difference is the

mode of operation. A multiprocessor system design is rather symmetrical consisting of a number of identical processor and memory components, controlled by one or more copies of the same operating system, which is responsible for a strict control of the task assignment to each processor. This is not true in distributed computing systems, where heterogeneity of the operating system as well as the hardware is quite common.

In addition, a DDBS is not a system where, despite the existence of a network, the database resides at only one node of the network. In this case, the problems of database management are no different from the problems encountered in a centralized database environment. The database is centrally managed by one computer system and all the requests are routed to that site. The only additional consideration has to do with transmission delays. It is obvious that the existence of a computer network or a collection of "files" is not sufficient to form a distributed database system.

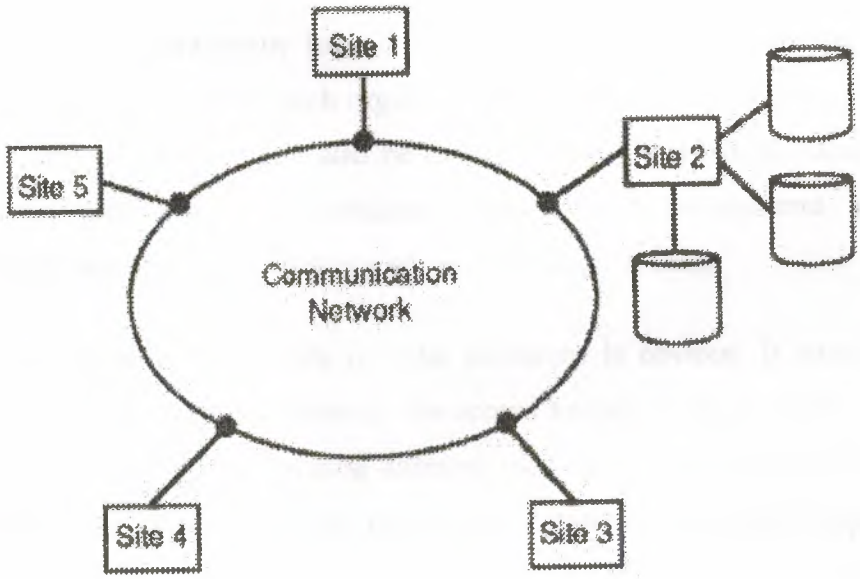

Figure 1.4: Central Database on a Network

At this point it might be helpful to look at an example of distributed database application that we can also use to clarify our subsequent discussions.

1.2 ADVANTAGES AND DISADVANTAGES OF DDBS

The distribution of data and applications has promising potential advantages. Note that these are potential advantages which the individual DDBSs aim to achieve. As such, they may also be considered as the objectives of DDBSs.

1.2.1 Advantages

Local Autonomy: Since data is distributed, a group of users that commonly share such data can have it placed at the site where they work, and thus have local control. This permits setting and enforcing local policies regarding the use of the data. There are studies [D'Oliviera, 1977] indicating that the ability to partition the author ity and responsibility of information management is the major reason many business organizations consider distributed information systems. This is probably the most important sociological development that we have witnessed in recent years with respect to the use of computers.

Of course, the local autonomy issue is more important in those organizations that are inherently decentralized. For such organizations, implementing the information system in a decentralized manner might also be more suitable. On the other hand, for those organizations with quite a centralized structure and management style, decentralization might not be an overwhelming social or managerial issue.

In distributed system, the validity of local autonomy is obvious. It would be quite absurd to have an environment where all the record keeping is done locally, as it would be if information were shared among different sites in a manual fashion (either by exchanging hard copies of reports, or by exchanging magnetic tapes, disks, floppies, etc.).

Improved Performance: Again, because the regularly used data is proximate to the users, and given the parallelism inherent in distributed systems, it may be possible to improve the performance of database accesses. On the one hand, since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases. On the other hand, data retrieved by a transaction

may be stored at a number of sites, making it possible to execute the transaction in parallel.

Let us assume that in our example the record keeping is done centrally at the world headquarters, with remote access provided to the other sites. This would require the transmission to New York of each request generated in Phoenix inquiring about the inventory level of an item. It would probably be impossible to withstand the low performance of such an operation.

Improved Reliability/Availability: If data is replicated so that it exists at more than one site, a crash of one of the sites, or the failure of a communication link making some of these sites inaccessible, does not necessarily make the data impossible to reach. Furthermore, system crashes or link failures do not cause total system inoperability. Even though some of the data may be inaccessible, the DDBS can still provide limited service.

Obviously, if the inventory information at both warehouses is replicated at both sites, the failure at one of the sites would not make the information inaccessible to the rest of the organization. If proper facilities are set up, it might even be possible to give users at the failed site access to the remote information.

Economics: It is possible to view this from two perspectives. The first is in terms of communication costs. If databases are geographically dispersed and the applications running against them exhibit strong interaction of dispersed data, it may be much more economical to partition the application and do the processing locally at each site. Here the trade-off is between telecommunication costs and data communication costs. The second viewpoint is that it normally costs much less to put together a system of smaller computers with the equivalent power of a single big machine. In the 1960s and early 1970s, it was commonly believed that it would be possible to purchase a fourfold powerful computer if one spent twice as much. This was known as Grosh's law. With the advent of minicomputers, and especially microcomputers, this law is considered invalid.

The case about lower communication costs can easily be demonstrated in the example we have been considering. It is no doubt much cheaper in the long run to

maintain a computer system at a site and keep data locally stored instead of having to incur heavy telecommunication costs for each request. The level of use when this becomes true can obviously change depending on the traffic patterns among sites, but it is quite reasonable to expect this to occur.

Expandability: In a distributed environment, it is much easier to accommodate increasing database sizes. Major system overhauls are seldom necessary; expansion can usually be handled by adding processing and storage power to the network. Obviously, it may not be possible to obtain a linear increase in "power," since this also depends on the overhead of distribution. However, significant improvements are still possible.

Share ability: Organizations that have geographically distributed operations normally store data in a distributed fashion as well. However, if the information system is not distributed, it is usually impossible to share these data and resources. A distributed database system therefore makes this sharing feasible.

1.2.2 Disadvantages

However, these advantages are offset by several problems arising from the distribution of the database.

Lack of Experience: General-purpose distributed database systems are not yet commonly used. What we have are either prototype systems or systems that are tailored to one application (e.g., airline reservations). This has serious consequences because the solutions that have been proposed for various problems have not been tested in actual operating environments.

Complexity: DDBS problems are inherently more complex than centralized database management ones, as they include not only the problems found in a centralized environment, but also a new set of unresolved problems. We discuss these new issues shortly.

Cost: Distributed systems require additional hardware (communication mechanisms, etc.), thus have increased hardware costs. However, the trend toward

decreasing hardware costs does not make this a significant factor. A more important fraction of the cost lies in the fact that additional and more complex software and communication may be necessary to solve some of the technical problems. The development of software engineering techniques (distributed debuggers and the like) should help in this respect.

Distribution of Control: This point was stated previously as an advantage of DDBSs. Unfortunately, distribution creates problems of synchronization and coordination (the reasons for this added complexity are studied in the next section). Distributed control can therefore easily become a liability if care is not taken to adopt adequate policies to deal with these issues.

Security: One of the major benefits of centralized databases has been the control it provides over the access to data. Security can easily be controlled in one central location, with the DBMS enforcing the rules. However, in a distributed database system, a network is involved which is a medium that has its own security requirements. It is well known that there are serious problems in maintaining adequate security over computer networks. Thus the security problems in distributed database systems are by nature more complicated than in centralized ones.

Difficulty of Change: Most businesses have already invested heavily in their database systems, which are not distributed. Currently, no tools or methodologies exist to help these users convert their centralized databases into a DDBS. Research in heterogeneous databases and database integration is expected to overcome these difficulties.

CHAPTER TWO

DISTRIBUTED SYSTEMS AND DISTRIBUTED SOFTWARE

2.1 CHARACTERISTIC OF DISTRIBUTED SYSTEMS

Distributed computer environments are based on distributed computer systems which consist of a set of processing components connected by a communication network. The software systems running on the various processing components exchange data through the communication network. This type of system is also called loosely coupled distributed system.

Processing nodes can be composed of several processors which share memory. This shared memory is used to exchange information by the software executed on such a node. This type of system is called a tightly coupled distributed system. Some advantages of distributed systems are below shown:

- Increased Performance

Performance is generally defined in terms of average response time and through put. If processing capability can be located where it is required the response time can be highly reduced. Data can be processed locally before it is sent to other nodes for further processing. This increases throughput.

- Increased reliability

Normally nodes in a distributed system can take over the tasks of other nodes which are currently out of order. This means that a distributed system continues its work with reduced performance but with little or no reduction of functionality

- Increased flexibility

Additional functionality can be added to a distributed system or the number of users can be permanently increased. A distributed system allows this system growth by simply adding more processing nodes.

2.2 PARALLEL OR CONCURRENT PROGRAMS

Parallel or concurrent programs are characterized by a set of statements interrelated by multiple control threads. Each sequence of statements executed by one or more control threads is called a process object (The term 'process' shall be used instead of 'process object' when it is clear from the context that we mean a process object).

The relationship between processes or threads and process objects is shown in the following figure.

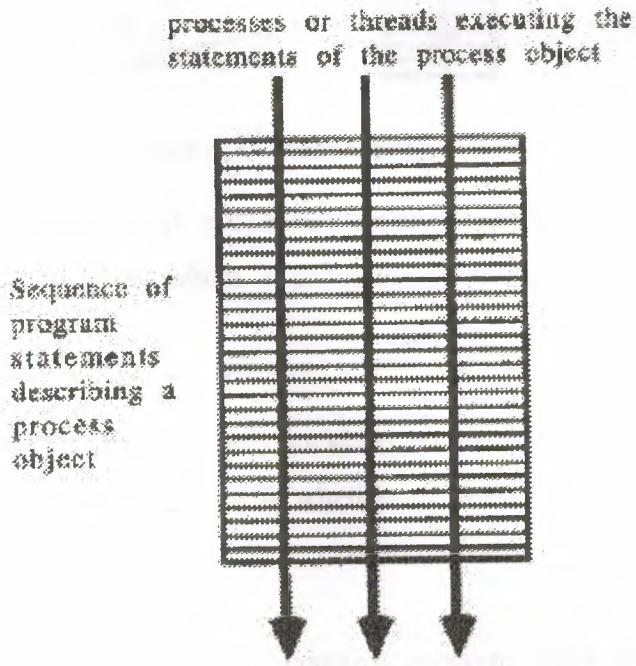

Figure 2.1: Process/Threads and Process Objects

The statements (operations) of the individual processes are executed overlapped or interleaved or both. If a single processor is multiplexed among several concurrent processes, the machine instructions of these processes can only be interleaved in time. For a certain time slice, the processor is assigned to a process in order to execute the statements of a process object. Assigning a processor to another process is called context switching. This type of concurrency is also called multitasking. The following figure shows an example of how a processor is shared between several processes.

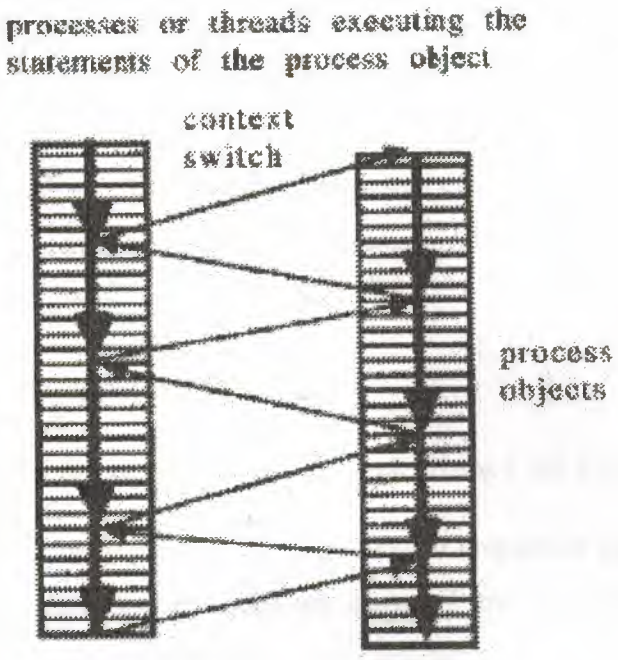

Figure 2.2: Multitasking

Machine instructions of processes running on different processors can be overlapped at each node at which a processor is available. These are distributed programs.

Concurrent or parallel programs are either interleaved, distributed, or both. For a programmer it is not necessary to know whether multitasking or a distributed system is used to run his program.

Normally the processes of a concurrent program share the resources such as processor, memory, disk, and databases, and if they cooperate in order to reach a common goal they exchange information and synchronize their activities.

Their are two reasons to structure a program in parallel executable process objects:

1. Fine grain parallelism is mainly used to accelerate large numerical computations. This type of parallels is often achieved by using vector processors and the pipelining of operations. It is mainly implemented by hardware.
2. Structural parallelism is used if the structure of the task to be performed is fundamentally parallel. The process objects are a very important concept for structuring

programs in certain application areas, e.g. operating systems, real time systems, and communication systems. Especially in real time systems which must react to external events, processes (objects) are used to achieve separation of the tasks /FAPA88/. Each process handles a related set of events and cooperates with other processes to achieve a common purpose. In order to cooperate, processes exchange information either via shared data or via messages.

2.3 NETWORKED COMPUTING

2.3.1 Network Structure and the Remote Procedure Call Concept

Network computing is characterized by several sequences of jobs, which arrive independently at various nodes. The jobs are designed and implemented more or less independently of each other and are only loosely coupled. The distributed system serves primarily as a resource-sharing network.

A very common example of resource sharing is the file server. All files are located on a dedicated node in a distributed system. Software components running on other nodes send their file access requests to the file server software. The file server executes these requests and returns the results (to the clients).

In addition to file servers many other kinds of servers such as print servers, compute servers, data base servers, and mail servers have been implemented As with the file server, clients send their requests to the appropriate server and receive the results for further processing. Servers process the requests from the various clients more or less independently of each other. The programs running on the clients can be viewed as being designed and developed independently of each other.

The following figure shows the concept of client/server system

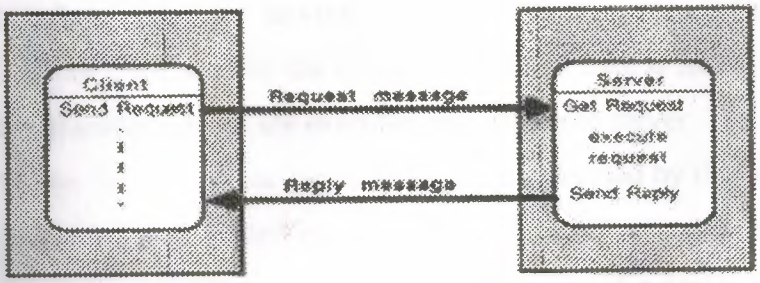

Figure 2.3: The Concept of Client/Server System

In client server systems, the clients represent the users of a distributed system and servers represent different operating system functions or a commonly used application.

The following figure shows a simple example of a client server system.

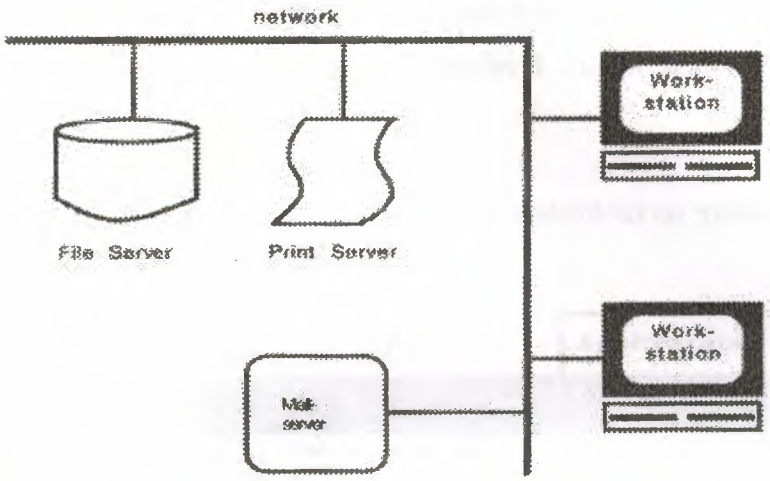

Figure 2.4: A Small Client/Server System

This system has a print server, a file server, and the clients (users) which run on workstations (WS) and personal computers (PC). The server software and the client software can run on the same type of computer. The different nodes are connected by a local area network.

From a user's point of view a client/server system can hardly be distinguished from a central system, e.g. a user cannot see whether a file is located on his local system or on a remote file server node. For the user the client/server system appears to be a very convenient and flexible central computing system. Mostly the user does not know whether a file is stored on his PC or on a file server. To the user, the storage capacity of the server appears to be a part of the PC storage capacity. Client/server systems are also very flexible. For a new application a specialized new server can be added e.g. data base systems run on specialized data base servers, which have short access times. Database applications are primarily controlled by the local client; all the data is stored at the data base server and special computations are executed by a compute server. The application program running on the client, calls the required functions provided by the servers. This is done mainly by way of remote procedure calls (RPC). An RPC resembles a procedure call except that it is used in distributed systems. The following is a description of how the RPC works. The program running on the client looks like a normal sequential

program. The services of a particular server are invoked via a remote procedure call. The caller of a remote procedure is stopped until the invoked remote procedure is finished and the server has provided the results to the calling client in the same way that parameters are returned by a procedure. The servers are used in the same way that library procedures are used. This means that remote procedure calls hide the distribution of the functions of the system even at the program level. The programmer does not need to concern himself with the system distribution.

The figure below shows the basic structure of a client/server system.

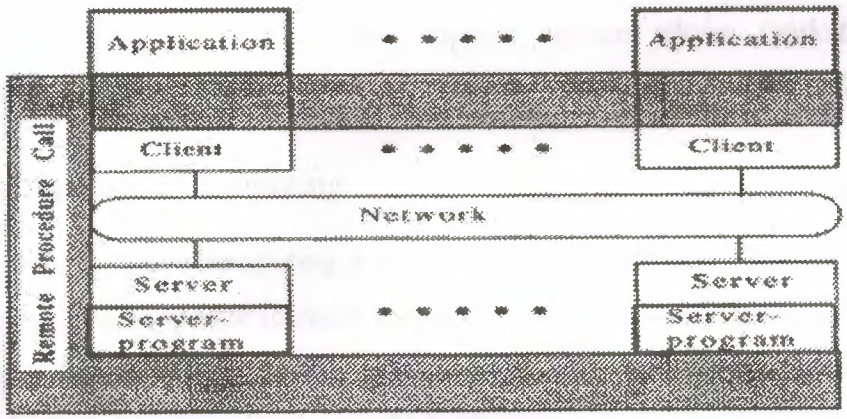

Figure 2.5: Remote Procedure Call Concept

2.3.2 Distributed Computing Environment (DCE)

The Distributed Computing Environment is a comprehensive integrated set of tools which supports network computing in a heterogeneous computing environment. This set of technologies has been selected by the Open Systems Foundation (OSF) to support the development of distributed applications for heterogeneous computer networks. The following figure shows the OSF DCE architecture.

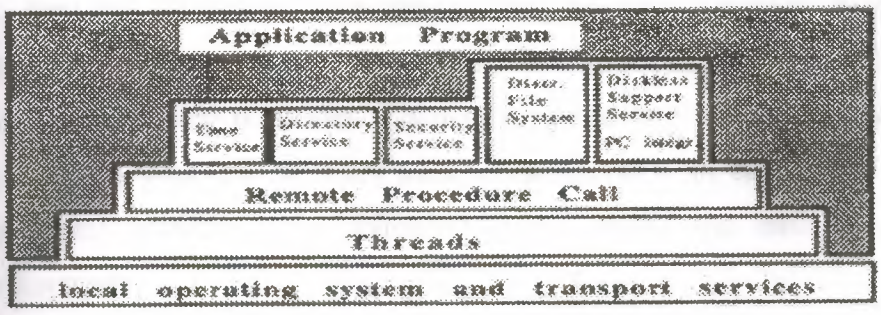

Figure 2.6: Architecture of OSFDCE

In the DCE client and server programs are executed by threads i.e. processes. Threads use an RPC in order to communicate with each other and binary semaphores and conditional variables for synchronization. In the DCE remote procedure calls are supported by directory services (DCE Call Directory Service) and security services (DCE Security Service). Directory services map logical names to physical addresses. If a client calls a particular service provided by a server, the directory service is used to find the appropriate server. The DCE security service provides features for secure communication and controlled access to resources. Distribute Time Service provides precise clock synchronization in a distributed system. This is required for event logging, error recovery, etc. The distributed file service allows the sharing of files across the whole system. Finally the diskless support service allows workstations to use background disk files on file servers as if they were local disks /SCHILL93/, /OSF92/.

2.3.3 Cooperative Computing

In cooperative computing a set of processes runs on several processing nodes. These processes cooperate to reach a common goal and together they form a distributed program. This is different from the client/server systems described above. In cooperative systems the processes, which comprise the distributed program are coupled very closely. This means that the closely coupled processes are executed on a loosely coupled system.

In cooperative systems, the distribution of computing capability is not hidden behind programming concepts. The different program sections running on different computers comprise a single program; but it can be seen at the programming level that the program sections are executed concurrently. These different program sections are also processes. Processes form a very important concept for central systems, client server systems and cooperative systems. If processes have to work together to perform their task, they must exchange data and synchronize their execution. Programming systems for concurrent systems contain communication and synchronization concepts. Cooperative programming resembles a human organization which works together to achieve a common goal. Its members must communicate with each other and must synchronize their activities. The following figure shows the basic structure of cooperative systems.

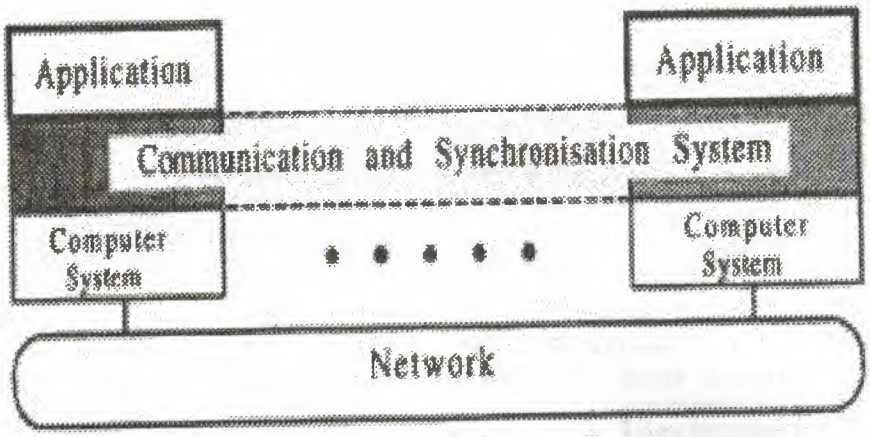

Figure 2.7: Structure of Cooperative Systems

Cooperative systems are mainly used for the automation of technical processes and the implementation of communication software, etc. Technical processes in the mostly part consist of several parallel activities, for example checking the level of a tank has to be done in parallel with controlling the rate of flow of a pump. Therefore the structure of technical process control software is very similar to the structure of the technical process to be controlled. For the automation of technical processes such as manufacturing control systems, the environment of the program, the technical process, is considered as a set of processes which interact with software processes. This means that several processes which can be implemented in different ways work together to perform their task.

2.4 COMMUNICATION SOFTWARE SYSTEMS

A communication system consists of a communication network and the communication software, which runs on the various processing nodes (referred to as host systems). The communication software provides a more or less convenient communication service for the application software. The application software on each node uses the communication service to exchange messages with the application software running on other nodes. The communication service is based on the underlying network (A network is usually made up of lines and several switching nodes although most local area networks do not contain switching nodes).

In order to provide a convenient communication service the communication software systems also exchange messages. This message exchange is based on the

simpler communication mechanism provided directly by the network. For example the network provides a communication service, which only allows the transfer of a single byte. The communication service provided by the communication software allows byte strings of a fixed or even an unlimited length to be sent or received. This can be implemented in the following way:

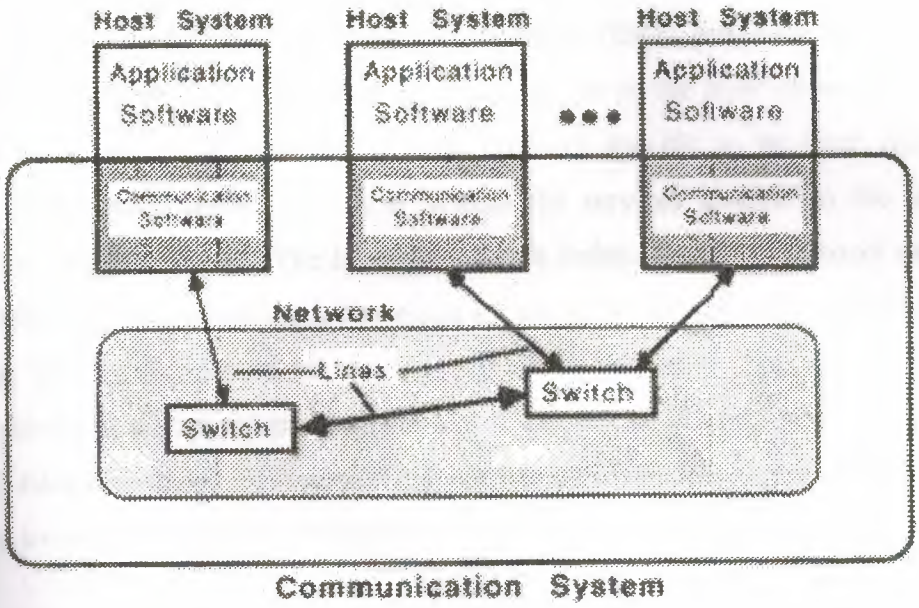

Figure 2.8: Structure of Communication Systems

The application software of a host system A wants to send a sequence of bytes to the application software of a host system B. The sequence of bytes is given to the communication system by the application system. The communication system on host system A sends a byte with the length of the byte string (the number of bytes) to the communication system on host system B. The communication system on host system B sends back an acknowledgement. This is a byte with a certain value. After the communication software on host system A has received the acknowledgement it starts to transfer the bytes of the byte string. When system B has received the number of bytes indicated in the first byte it again sends an acknowledgement. After sending the acknowledgement, the communication software on host system B gives the received byte string to the application software.

This communication sequence which implements the transfer of a byte string is just a simplistic illustration of what communication software can do.

As the example above shows, the communication between the communication software systems follows well-defined rules. These rules are called protocols. The need to provide convenient communication services for the application software leads to software communication protocols, which can be extremely complex and must be organized in layers. Each layer offers an improved communication service to the layer above. The widely used reference model for Open Systems Interconnection (OSI) defined by the International Standard Organization (ISO) proposes seven protocol layers /IS07498/. Each layer provides a certain service to the layer above. The service provided by a layer is implemented by the protocol specific to its layer and by the services of the layer below. In a host system the services specific to the layer are realized by protocol entities. The layer protocol is defined between protocol entities of the same layer. These exchange information by using the service of the layer below. In each host system there must be at least one entity per layer. The set of entities of different layers in a host system is called a protocol stack. The implementation of these protocol stacks is called communication software. Communication software has the following execution properties /DROB86/:

- interleaved execution of several entities on the same system
- distributed execution of entities of the same layer on different systems.

Interleaved and distributed computations are usually modeled as systems of parallel processes. Processes executing in parallel normally have to exchange information if they are to cooperate in solving a common task. One or more processes model entities. Using or providing a service means exchanging information with processes representing entities of the layer below or above. The figure above shows the structure of communication software systems based on the ISO/OSI reference model. Protocol stacks in the different host systems are implemented independently of each other and are embedded in the communication systems. This means that the implementation of a communication system to support communication in a distributed program is itself a distributed program.

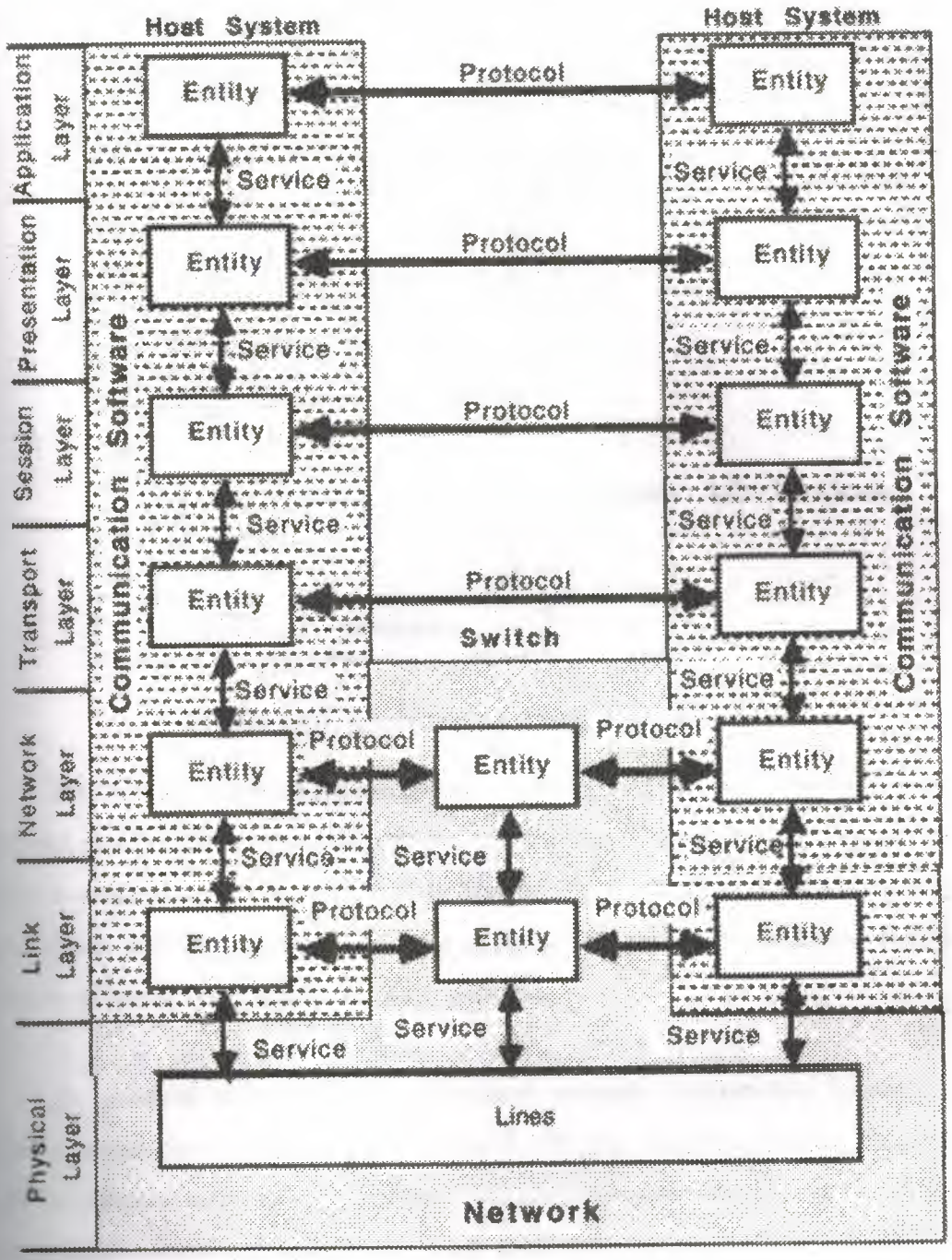

Figure 2.9: Structure of Communication Software

2.4.1 Technical Process Control Software Systems

Another important example of cooperative computing is a distributed technical process control system.

The basic structure of technical systems controlled by computer systems is shown in the following figure /NEHM84/.

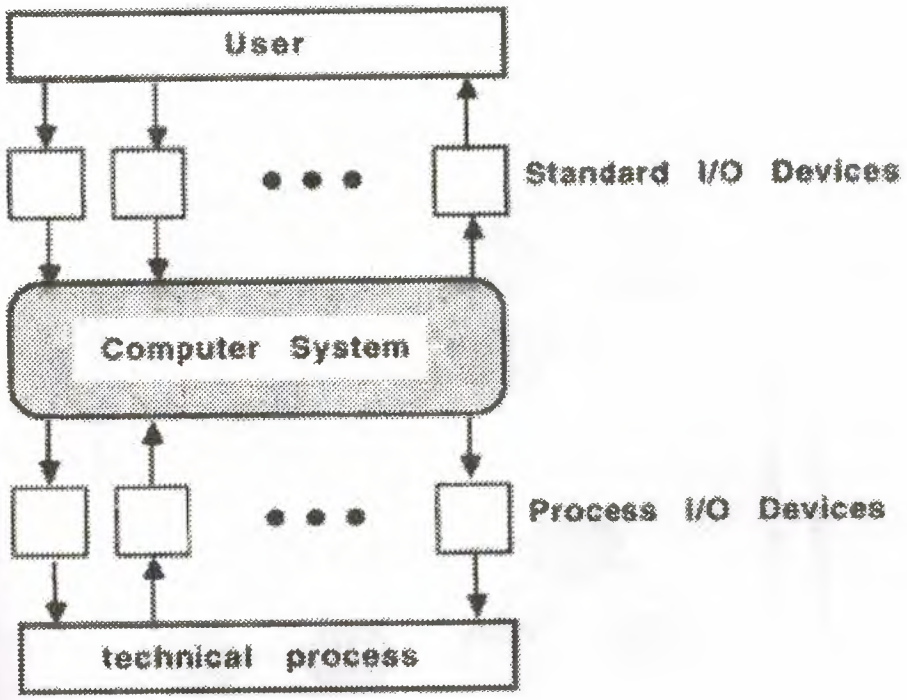

Figure 2.10: Structure of Process Control Systems

The communication between computer systems and technical systems must meet hard real time requirements, whereas the communication with the user is more or less dialogue-oriented with less emphasis on time conditions (except in the case emergency signals such as fire alarms). For the sake of simplicity, we will focus on the relationship between technical systems and real-time computer systems.

A technical system consists of several mutually independent functional units which communicate via appropriate interfaces with the computer system. Therefore the real time program must react to several simultaneous inputs. This implies the structuring of a process control software system that takes into account a number of processes. Each process handles a certain group of signals.

The basic requirement for a process control software system is the capability to follow the changes of the technical system as fast as possible. The information in the process control software must be as close as possible to the state of the technical system. The easiest way to achieve this is to design a process for each interface element. This leads to the software system structure shown in the following figure /NEHM84/.

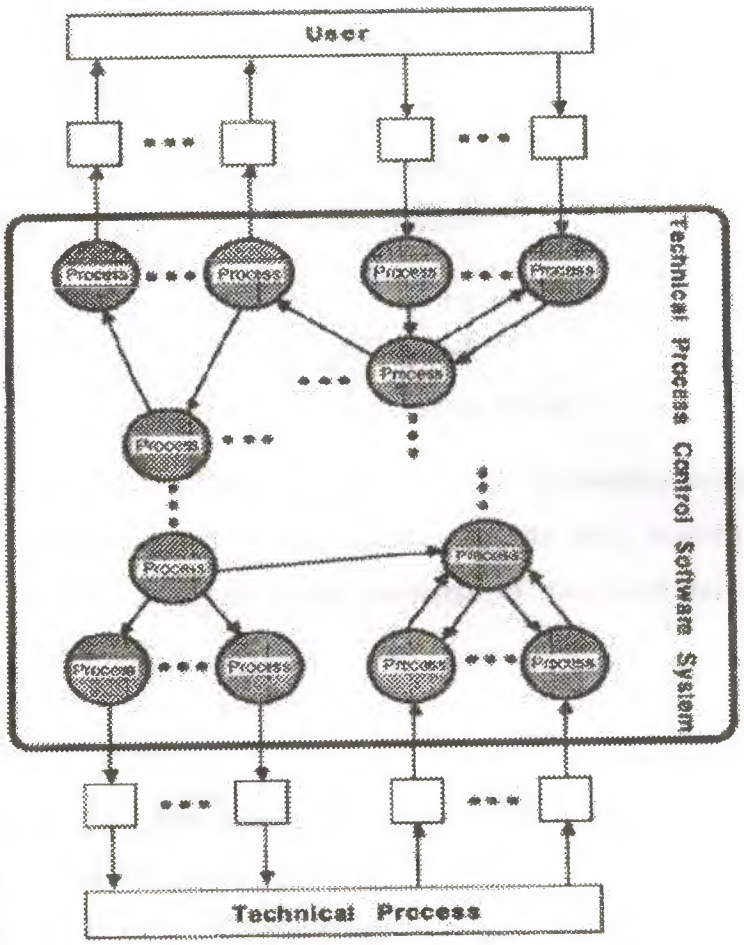

Figure 2.11: Structure of Process Control Software

Software system processes can run on a single centralized system or can be distributed over several computer systems. In the latter case it is possible to locate the computers close to the device or the plant being controlled. The main advantages of distributed solutions are:

- reduction of wiring costs
- faster response
- easier development and maintenance
- a higher degree of fault tolerance

2.4.2 Electronic Data Interchange (EDI)

Electronic Data Interchange (EDI) is the computer-to-computer exchange of inter and intra company technical and business data, based on the use of standards DIGIT90/ (see figure below of the EDI business model).

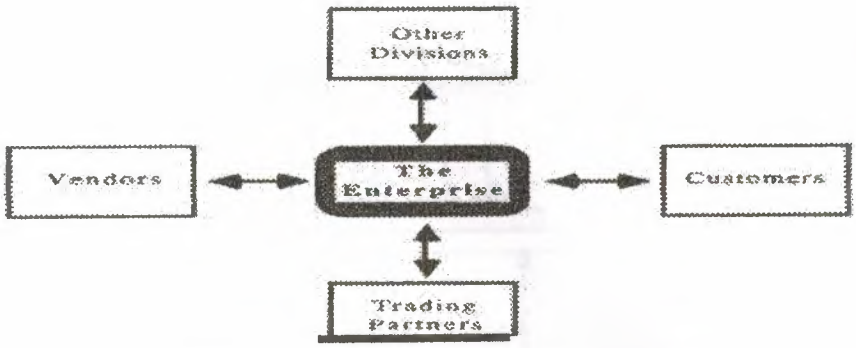

Figure 2.12: EDI Business Model

These data can be structured or unstructured. Exchanging unstructured data follows specific communication standards although the data content is not in a structured format. More important is the exchange of structured data. Examples of structured data exchange are:

- Trade Data Interchange

This type of EDI document exchange is mainly used to automate business processes. Examples of trade data interchanges include a request for quotation (RfQ), purchase orders, purchase order acknowledgements, etc. Each company and industry has its own requirements for the structure and contents of these documents. A number of specific industry and national bodies have been formed with the intention of standardizing the format and content of messages. For the chemical industry CEFIC is the EDI standard and for the auto industry the related EDI standard is called ODETTE. The standard defined by CCITT is called EDIFACT. In order to exchange EDIFACT documents very often the CCITT E-Mail standard X.400 is recommended /HILL90/.

- Electronic Funds Transfer

Payment against invoices, electronic point of sale (EPOS) and clearing systems are examples of electronic funds transfer.

- Technical Data Interchange

Improvement in technical communication can play a key role in determining the success of a project. There is a growing demand from traders for communication between their CAD (computer aided design) workstation and the workstations of important vendors.

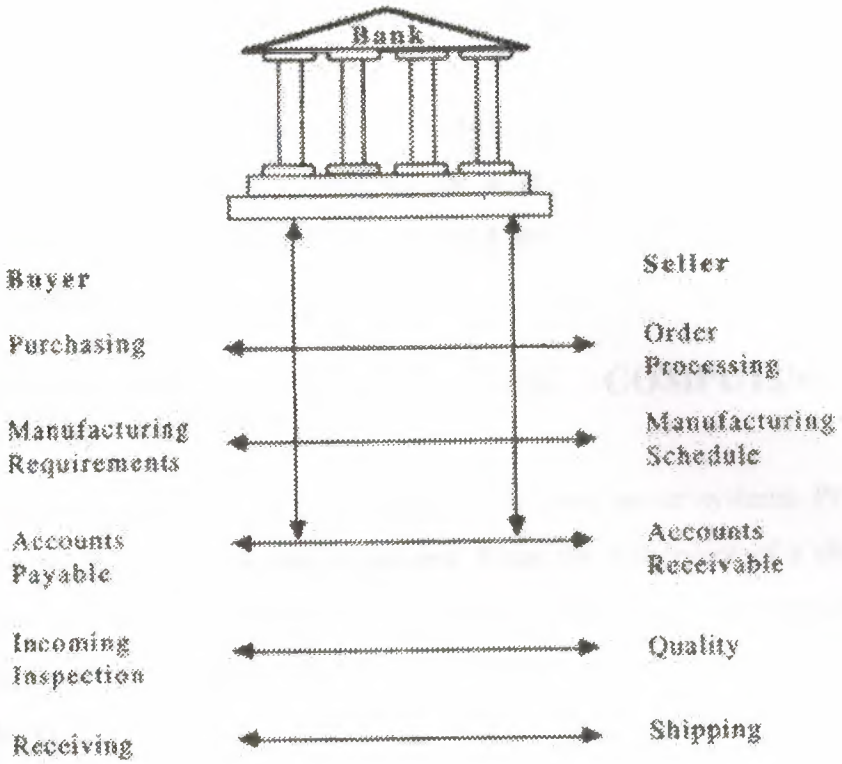

Figure 2.13: EDI in a Business Process

2.4.3 Groupware

In organizations people work together to reach a common goal. The formal interaction between members of an organization is described by structures and procedures. Additionally there exist informal interactions which are very important. Both types of interactions can and should be supported by computers. Computer Supported Cooperative Work (CSCW) deals with the study and development of computer systems called groupware, which purpose it is to facilitate these formal and informal interactions. CSCW projects can be classified into four types namely:

1. Groups which are not geographically distributed and require common access in real time Examples: presentation software, group decision systems.
2. Groups which are geographically distributed and require common access in Real time Examples: video conferencing, screen sharing.
3. Asynchronous collaboration among people who are geographically distributed. Examples: notes conferences, joint editing.

4. Asynchronous collaboration among people who are not geographically distributed Examples: project management, personal time schedule management

Groupware requires computers connected by a network. Thus groupware systems are distributed systems. Members of a group share data and exchange messages. Therefore groupware software systems are combinations of network and cooperative computing.

2.5 COMBINATION OF NETWORK COMPUTING AND COOPERATIVE COMPUTING

Cooperative computing can be combined with client server systems. Processes in a distributed system can have access to servers. From the standpoint of a client server system the processes of a cooperative system can be considered as client processes. In a technical process control software system a process can collect data from the technical process. This data is stored in a file located on a file server node. The following figure shows an example of a combination of a cooperative and a client/server system. Process A, Process B and Process C form a cooperative software system. Process B and Process C use the file server. This means that process B and process C are clients of the file server.

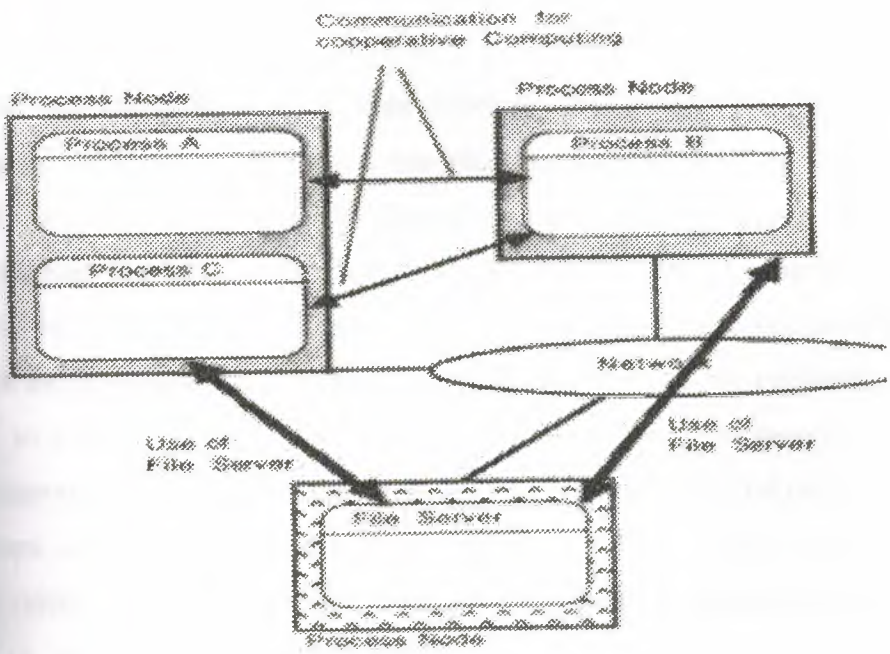

Figure 2.14: Combination of Cooperative and Client Server System

CHAPTER THREE

ARCHITECTURE OF DBMS

3.1 TRANSPARENCIES IN A DISTRIBUTED DBMS

Transparency in a distributed DBMS refers to separation of the higher-level semantics of a system from lower-level implementation issues. In other words, a transparent system "hides" the implementation details from users. The advantage of a fully transparent DBMS is the high level of support that it provides for the development of complex applications. It is obvious that we would like to make all DBMSs (centralized or distributed) fully transparent. In fact, we have alluded to this under the topic of data independence, which is one form of transparency. In the remainder of this section we consider the various forms of transparency that a designer aims to provide within centralized or distributed DBMS.

3.1.1 Data Independence

Data independence is a fundamental form of transparency that we look for within a DBMS. It is also the only type that is important within the context of a centralized DBMS. To reiterate the definition given data independence refers to the immunity of user applications to changes in the definition and organization of data, and vice versa.

As we will see in Section 4.2, data definition can occur at two levels. At one level the logical structure of the data is specified, and at the other level the physical structure of the data is defined. The former is commonly known as the schema definition, whereas the latter is referred to as the physical data description. We can therefore talk about two types of data independence: logical data independence and physical data independence. Logical data independence refers to the immunity of user applications to changes in the logical structure of the database. In general, if a user application operates on a subset of the attributes of a relation, it should not be affected later when new attributes are added to the same relation. For example, let us consider the engineer relation discussed. If a user application deals with only the address fields of this relation (it might be a simple mailing program), the later additions to the relation of say, skill, would not and should not affect the mailing application.

Physical data independence deals with hiding the details of the storage structure from user applications. When a user application is written, it should not be concerned with the details of physical data organization. The data might be organized on different disk types, parts of it might be organized differently (e.g., random versus indexed sequential access) or might even be distributed across different storage hierarchies (e.g., disk storage and tape storage). The application should not be involved with these issues since, conceptually; there is no difference in the operations carried out against the data. Therefore, the user application should not need to be modified when data I organizational changes occur with respect to these issues. Nevertheless, it is common knowledge that these changes may be necessary for performance considerations.

Of course, data independence is more of a goal than a standard feature commonly provided by most of today's DBMSs. Some commercial products provide better data independence than others. Specifically, most of the microcomputer DBMSs do not provide high levels of data independence. Adding a new attribute to a relation (i.e., logical data independence) very often requires unloading the database, changing the relation definition, and then reloading the database.

3.1.2 Network Transparency

In centralized database systems, the only available resource that needs to be shielded from the user is the data (i.e., the storage system). In a distributed database management environment, however, there is a second resource that needs to be managed in much the same manner: the network. Preferably, the user should be protected from the operational details of the network. Furthermore, it is desirable to hide even the existence of the network, if possible. Then there would be no difference between database applications that would run on a centralized database and those that would run on a distributed database. This type of transparency is referred to as network transparency or distribution transparency. One can consider network transparency from the viewpoint of either the services provided or the data. From the former perspective, it is desirable to have uniform means by which services are accessed. Tb give an example, let us talk for the moment not at the database level but at the operating system level in a network environment. If we want to copy a file, the command needed should be the same whether the file is being copied within one machine or across two machines connected by the network. Unfortunately, however, most commercially available

operating systems that run on networks do not provide this transparency. For example, the UNIX1 command for copying in one machine is

cp <source file> <target file>

Whereas the same command, if the source and the target files are on different machines, takes the form

rcp <machine _ name: source file> <machine _ name :target file>

Note how it is now necessary to name the machine on which the file resides and to use a different operating system command to perform the copy function. If the same discussion is carried over to the database level, we would see that different user interfaces (i.e., query languages and data manipulation languages) need to be designed for both centralized and distributed database environments. Clearly, this is not very desirable.

The example above demonstrates two things: location transparency and naming transparency (or the lack of these). Location transparency refers to the fact that the command used is independent of both the location of the data and the system on which an operation is carried out. Naming transparency means that a unique name is provided for each object in the database. It is obvious that in a system such as the one described above, the task of providing unique names for different objects falls on the user rather than the system. The way the system handles naming transparency is by requiring the user to embed the location name (or an identifier) as part of the object name.

It is unfortunate that some distributed database systems do indeed embed the location names within the name of each database object. Furthermore, they require the user to specify the full name for access to the object. Obviously, it is possible to set up aliases for these long names if the operating system provides such a facility. However, user-defined aliases are not real solutions to the problem in as much as they are attempts to avoid addressing them within the distributed DBMS. The system, not the user, should be responsible for assigning unique names to objects and for translating user-known names to these unique internal object names.

Besides these semantic considerations, there is also a very pragmatic problem associated with embedding location names within object names. Such an approach makes it very difficult to move objects across machines for performance optimization or other purposes. Every such move will require users to change their access names for the affected objects, which is clearly undesirable.

3.1.3 Replication Transparency

The issue of replicating data within a distributed database is discussed in quite some detail in. At this point, let us just mention that for performance, reliability, and availability reasons, it is usually desirable to be able to distribute data in a replicated fashion across the machines on a network. Such replication helps performance since diverse and conflicting user requirements can be more easily accommodated. For example, data that is commonly accessed by one user can be placed on that user's local machine as well as on the machine of another user with the same access requirements. This increases the locality of reference. Furthermore, if one of the machines fail, a copy of the data is still available on another machine on the network. Of course, this is a very simpleminded description of the situation. In fact, the decision as to whether to replicate or not, and how many copies of any database object to have, depends to a considerable degree on user applications. Note that replication causes problems in updating databases. Therefore, if the user applications are predominantly update oriented, it may not be a good idea to have too many copies of the data. As this discussion is the subject matter, we will not dwell further here on the pros and cons of replication.

Assuming that data is replicated, the issue related to transparency that needs to be addressed is whether the users should be aware of the existence of copies or whether the system should handle the management of copies and the user should act as if there is a single copy of the data (note that we are not referring to the placement of copies, only their existence). From a user's perspective the answer is obvious. It is preferable not to be involved with handling copies and having to specify the fact that a certain action can and/or should be taken on multiple copies. From a systems point of view, however, the answer is not that simple.

3.1.4 Fragmentation Transparency

The final form of transparency that needs to be addressed within the context of a distributed database system is that of fragmentation transparency. We discuss and Justify the fact that it is commonly desirable to divide each database relation into smaller fragments and treat each fragment as a separate database object (i.e., another relation). This is commonly done for reasons of performance, availability, and reliability. Furthermore, fragmentation can reduce the negative effects of replication.

Each replica is not the full relation but only a subset of it; thus less space is required and fewer data items need be managed.

When database objects are fragmented, we have to deal with the problem of handling user queries that were specified on entire relations but now have to be per formed on sub relations. In other words, the issue is one of finding a query processing strategy based on the fragments rather than the relations, even though the queries are specified on the latter. Typically, this requires a translation from what is called a global query to severe fragment queries. Since the fundamental issue of dealing with fragmentation transparency is one of query processing, we defer the discussion of techniques by which this translation can be performed.

3.1.5 Provide Transparency

It is possible to identify three distinct layers at which the services of transparency can be provided. It is quite common to treat these as mutually exclusive means of providing the service, although it is more appropriate to view them as complementary.

We could leave the responsibility of providing transparent access to data resources to the access layer. The transparency features can be built into the user language, which then translates the requested services into required operations. In other words, the compiler or the interpreter takes over the task and no transparent service is provided to the implementer of the compiler or the interpreter.

The second layer at which transparency can be provided is the operating system level. State-of-the-art operating systems provide some level of transparency to system users. For example, the device drivers within the operating system handle the minute details of getting each piece of peripheral equipment to do what is requested. The typical computer user, or even an application programmer, does not normally write device drivers to interact with individual peripheral equipment; that operation is transparent to the user.

Providing transparent access to resources at the operating system level can obviously be extended to the distributed environment, where the management of the

network resource is taken over by the distributed operating system. This is a good level at which to provide network transparency if it can be accomplished. The unfortunate aspect is that not all commercially available distributed operating systems provide a reasonable level of transparency in network management.

The third layer at which transparency can be supported is within the DBMS. In such a case one might talk about different modes of operation. In database machines, for example, the DBMS generally does not expect any transparent service from the operating system; in fact, there is no identifiable operating system other than a monitor and some device drivers. The DBMS acts as the integrated operating and database management system. A more typical environment is the development of a DBMS on a general-purpose computer running some operating systems. In this type of environment, the transparency and support for database functions provided to the DBMS designers is minimal and typically limited to very fundamental operations for performing certain tasks. It is the responsibility of the DBMS to make all the necessary translations from the operating system to the higher-level user interface. This mode of operation is the most common method today. There are, however, various problems associated with leaving the task of providing full transparency to the DBMS. These have to do with the interaction of the operating system with the distributed DBMS.

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. Specifically, it should be noted that the interface to a distributed DBMS does not need to be a programming language but can be a graphical user interface, a natural language interface, and even a voice system.

A hierarchy of these transparencies is shown in Figure 3.1. It is not always easy to delineate clearly the levels of transparency, but such a figure serves an important

instructional purpose even if it is not fully correct. To complete the picture we have added a "language transparency" layer, although it is not discussed in this chapter. With this generic layer, users have high-level access to the data (e.g., fourth-generation languages, graphical user interfaces, natural language access, etc.).

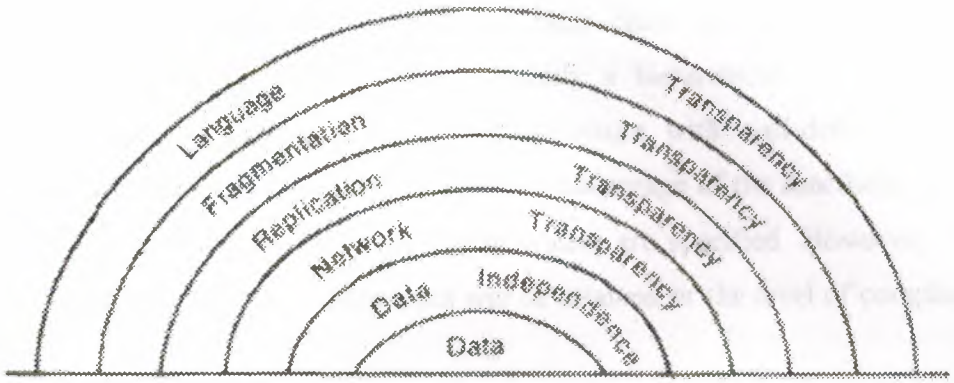

Figure 3.1: Layers of Transparency

3.2 DBMS STANDARDIZATION

In this section we discuss the standardization efforts related to DBMSs because of the close relationship between the architecture of a system and the reference model of that system, which is developed as a precursor to any standardization activity. For all practical purposes, the reference model can be thought of as an idealized architectural model of the system. It is defined as "a conceptual framework whose purpose is to divide standardization work into manageable pieces, and to show at a general level how these pieces are related with each other". Even though there is some controversy as to the desirability of standardization of DBMSs, it is a useful activity to the extent that it can establish uniform interfaces to the users and to other higher-level software developers. A reference model (and therefore system architecture) can be described according to three different approaches:

1. Based on components. The components of the system are defined together with the interrelationships between components. Thus a DBMS consists of a number of components, each of which provides some functionality. Their orderly and well-defined interaction provides total system functionality. This is a desirable approach if the ultimate objective is to design and implement the system under consideration. On the other hand, it is difficult to determine the functionality of a system by examining its

components. The DBMS standard proposals prepared by the Computer Corporation of America for the National Bureau of Standards ([CCA, 1980] and [CCA, 1982]) fall within this category.

2. Based on functions. The different classes of users are identified and the functions that the system will perform for each class are defined. The system specifications within this category typically specify a hierarchical structure for user classes. This results in hierarchical system architecture with well-defined interfaces between the functionalities of different layers. The advantage of the functional approach is the clarity with which the objectives of the system are specified. However, it gives very little insight into how these objectives will be attained or the level of complexity of the system.

3. Based on data. The different types of data are identified, and an architectural framework is specified which defines the functional units that will realize or use data according to these different views. Since data is the central resource that a DBMS manages, this approach is claimed to be the preferable choice for standardization activities [DAFTG, 1986]. The advantage of the data approach is the central importance it associates with the data resource. This is significant from the DBMS viewpoint since the fundamental resource that a DBMS manages is data. On the other hand, it is impossible to specify an architectural model fully unless the functional modules are also described. The ANSI/SPARC discussed in the next section belongs in this category.

Even though three distinct approaches are identified, one should never lose sight of the interplay among them. As indicated in a report of the Database Architecture Framework "Task Group of ANSI [DAFTG, 1986], all three approaches need to be used together to define an architectural model, with each point of view serving to focus our attention on different aspects of an architectural model.

3.3 ANSI / SPARC ARCHITECTURE

Two important events in the late 1960s and early 1970s influenced the standardization activities in database management. The Database Task Group (DBTG) of the C of DASYL Systems Committee issued two reports, one providing a survey of DBMSIB, and the second describing the features of a network DBMS. The second

event is the publication of Cod's initial papers on the relational data model. The existence of two alternative data models competing for dominance created considerable discussion not only of the merits of each, but also of the features of the next generation DBMSs.

In late 1972, the Computer and Information Processing Committee (X3) of the American National Standards Institute (ANSI) established a Study Group on Database Management Systems under the auspices of its Standards Planning and Requirements Committee (SPARC). The mission of the study group was to study the feasibility of setting up standards in this area, as well as determining which aspects should be standardized if it was feasible. The study group issued its interim report in 1975 and its final report in 1977. The architectural framework proposed in these reports came to be known as the 'ANSI/SPARC architecture," its full title being 'ANSI/X3/SPARC DBMS Framework." The study group proposed that the interfaces be standardized, and defined an architectural framework that contained 43 interfaces, 14 of which would deal with the physical storage subsystem of the computer and therefore not be considered essential parts of the DBMS architecture.

One of alternative approaches to standardization, the ANSI/SPARC architecture is claimed, to be based on the data organization. It recognizes three views of data: the external view, which is that of the user, who might be a programmer; the internal view, that of the system or machine; and the conceptual view, that of the enterprise. For each of these views, an appropriate schema definition is required. Figure 3.2 depicts the ANSI/SPARC architecture from the data organization perspective.

At the lowest level of the architecture is the internal view, which deals with the physical definition and organization of data. The location of data on different storage devices and the access mechanisms used to reach and manipulate data are the issues dealt with at this level. At the other extreme is the external view, which is concerned with how users view the database. An individual user's view represents the portion of the database that will be accessed by that user as well as the relationships that the user would like to see among the data. A view can be shared among a number of users, with the collection of user views making up the external schema. In between these two extremes is the conceptual schema, which is an abstract definition of the database. It

the "real world" view of the enterprise being modeled in the database. As such, it is supposed to represent the data and the relationships among data without considering the requirements of individual applications or the restrictions of the physical storage media. In reality, however, it is not possible to ignore these requirements completely, due to performance reasons.

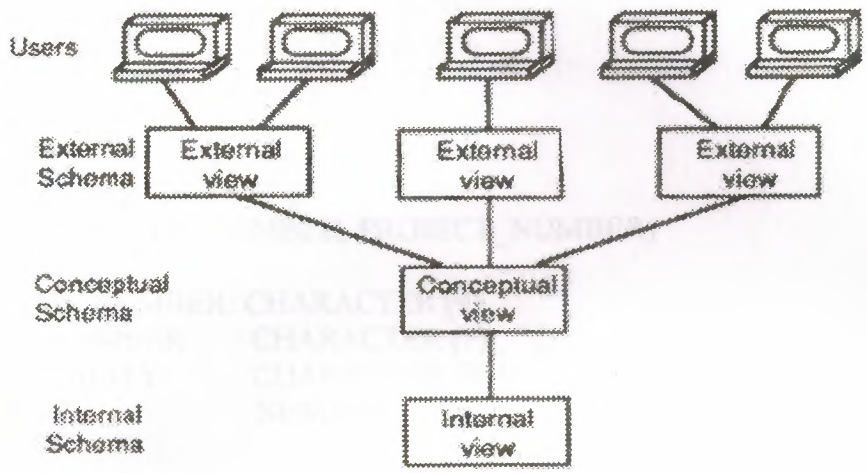

Figure 3.2: The ANSI/SPARC Architecture

These three levels is accomplished by mappings that specify how a definition at one can be obtained from a definition at another level.

Example:

Let us consider the engineering database example we have been using and indicate how it can be described using a fictitious DBMS that conforms to the ANSI/SPARC architecture. Remember that we have four relations: E, S, J, and G. The conceptual schema should describe each relation with respect to its attributes and its key. The description might look like the following: 2

```
RELATION EMPLOYEE [
KEY = {EMPLOYEE_NUMBER}
ATTRIBUTES = {
EMPLOYEE_NUMBER: CHARACTER (9)
EMPLOYEE_NAME    : CHARACTER (15)
TITLE            : CHARACTER (10)
}
]
RELATION TITLE.SALARY [
```

```
KEY = {TITLE}
ATTRIBUTES = {
TITLE              : CHARACTER (10)
SALARY             : NUMERIC (6)
}
]
RELATION PROJECT [
KEY = {PROJECT.NUMBER}
ATTRIBUTES = {
PROJECT.NUMBER   : CHARACTER (7)
PROJECT_NAME     : CHARACTER (20)
BUDGET             : NUMERIC (7)
}

RELATION ASSIGNMENT [
KEY = {EMPLOYEE_NUMBER, PROJECT_NUMBER}
ATTRIBUTES = {
EMPLOYEE_NUMBER: CHARACTER (9)
PROJECT.NUMBER   : CHARACTER (7)
RESPONSIBILITY   : CHARACTER (IO)
DURATION           : NUMERIC (3)
}
]
```

We used more descriptive names for the relations and the attributes. This is not the essential issue; a more important aspect is that these names can be different at all three levels, as we demonstrate below.

At the internal level, the storage details of these relations are described. Let us assume that the EMPLOYEE relation is stored in an indexed file, where the index is defined on the key attribute (i.e., the EMPLOYEE-NUMBER) called EMINX.3 Let us also assume that we associate a HEADER field, which might contain flags (delete, update, etc.) and other control information. Then the internal schema definition of the relation may be as follows:

```
INTERNAL_REL EMP [
INDEX       ON E# CALL EMINX
FIELD = {
E#          : BYTE(9)
E:NAME      : BYTE(15)
TIT         : BYTE (10)
}
]
```

We have used similar syntaxes for both the conceptual and the internal descriptions. This is done for convenience only and does not imply the true nature of languages for these functions.

Finally, let us consider the external views, which we will describe using SQL notation. We consider two applications: one that calculates the payroll payments for engineers, and a second that produces a report on the budget of each project.4 Notice that for the first application, we need attributes from both the EMPLOYEE and the TITLE-SALARY relations. In other words, the view consists of a join, which can be defined as

```
CREATE   VIEW   PAYROLL (ENO, ENAME, SAL)
AS       SELECT EMPLOYEE.EMPLOYEE_NUMBER,
EMPLOYEE . EMPLOYEE_NAME,
TITLE_SALARY.SALARY
FROM     EMPLOYEE, TITLE_SALARY
WHERE    EMPLOYEE.TITLE=TITLE_SALARY.TITLE
```

The second application is simply a projection of the PROJECT relation, which can be specified as,

```
CREATE   VIEW   BUDGET (PNAME, BUD)
AS       SELECT PROJECT.NAME, BUDGET
FROM PROJECT
```

The investigation of the ANSI/SPARC architecture with respect to its functions results in a considerably more complicated view, the square boxes represent processing functions, whereas the hexagons are administrative roles. The arrows indicate data, command, program, and description flow, whereas the "I" shaped bars on them represent interfaces.

The major component that permits mapping between different data organizational views is the data dictionary/directory (depicted as a triangle), which is a meta database. It should at least contain schema and mapping definitions. It may also contain usage statistics, access control information, and the like. It is clearly seen that the data dictionary/directory serves as the central component in both processing different schemas and in providing mappings among them.

In addition to these three classes of administrative user defined by the roles, there are two more, the application programmer and the system programmer. Two more user classes can be defined, namely casual users and novice end users. Casual users occasionally access the database to retrieve and possibly to update information. Such users are aided by the definition of external schemas and by an easy-to-use query language. Novice users typically have no knowledge of databases and access information by means of predefined menus and transactions (e.g., banking machines).

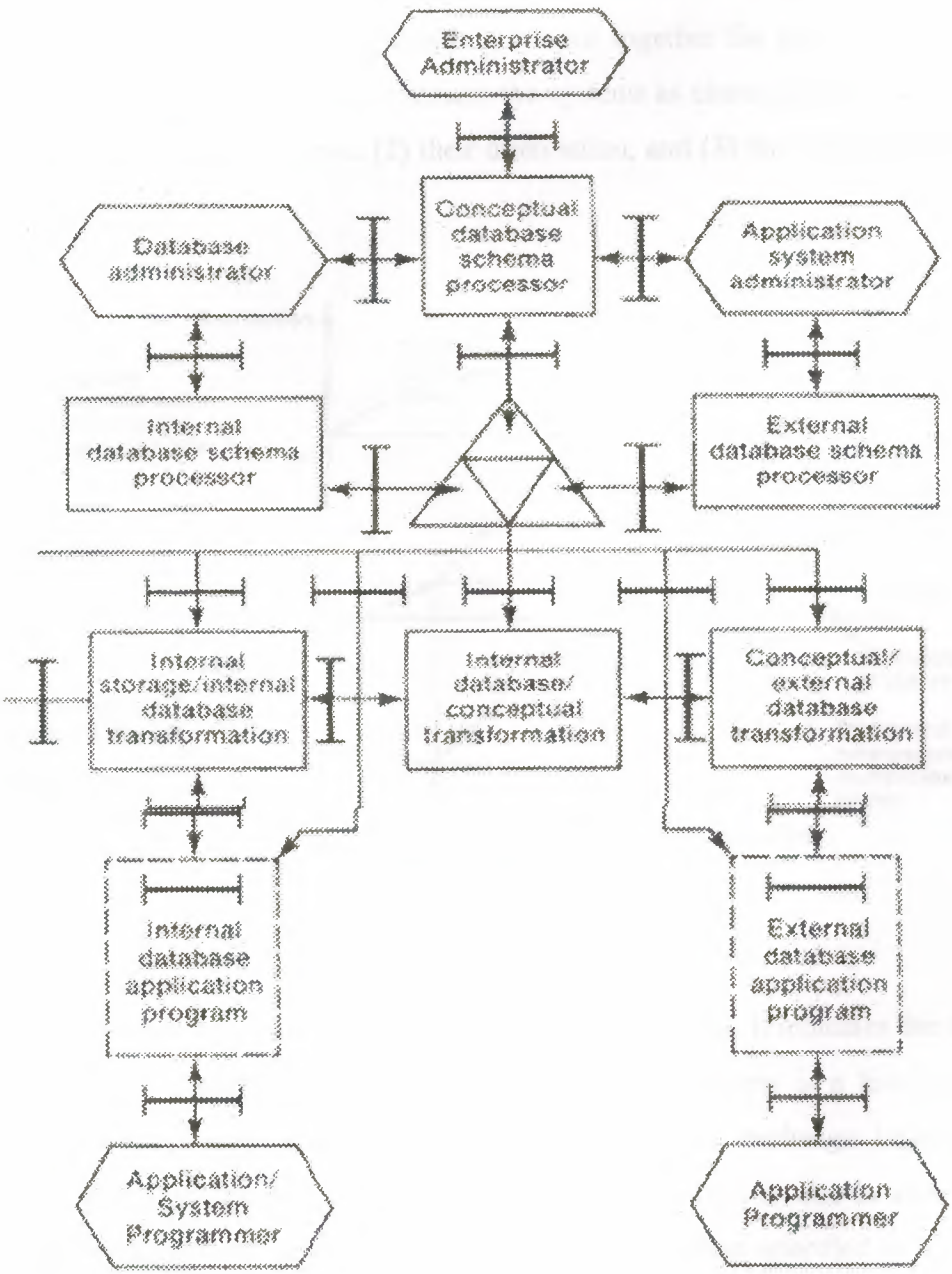

Figure 3.3: Partial Schematic of the ANSI/SPARC Architectural Model

3.4 ARCHITECTURAL MODELS FOR DISTRIBUTED DBMSS

The intuitive and logical nature of the ANSI/SPARC architecture has prompted many researchers to investigate ways of extending it to the distributed environment. The proposals range from simple extensions, such as that described by [Mohan and Yeh, 1978], to very complicated ones, such as Schreiber's model [Schreiber, 1977], and anything in between. In this book we use a simple extension of the ANSI/SPARC architecture.

Before discussing the specific architecture, however, we need to discuss the possible ways in which multiple databases may be put together for sharing by multiple DBMSs. We use a classification that organizes the systems as characterized with respect to **(1)** the autonomy of local systems, **(2)** their distribution, and **(3)** their heterogeneity.

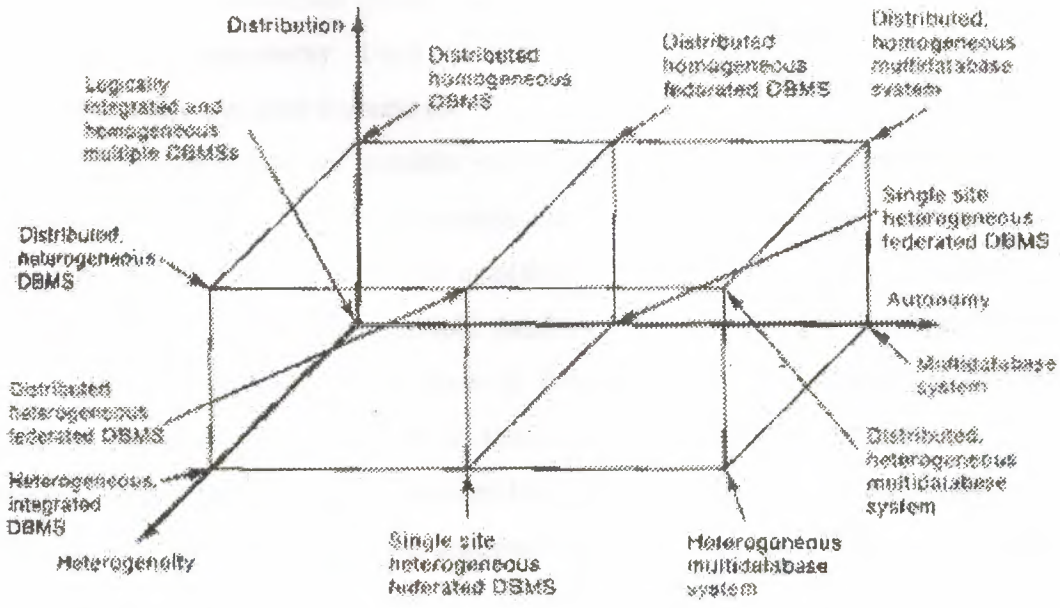

Figure 3.4 : DBMS Implementation Alternatives

Autonomy refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently. Autonomy is a function of a number of factors such as whether the component systems exchange information, whether they can independently execute transactions, and whether one is allowed to modify them. Requirements of an autonomous system have been specified in a variety of ways. For example, lists these requirements as follows:

1. The local operations of the individual DBMSs are not affected by their participation in the multi database system.

2. The manner in which the individual DBMSs process queries and optimize them should not be affected by the execution of global queries that access multiple databases.

3. System consistency or operation should not be compromised when individual DBMSs join or leave the multi-database confederation.

On the other hand, [Du and Elmagarmid, 1989] specifies the dimensions of autonomy as:

1. Design autonomy: Individual DBMSs are free to use the data models and transaction management techniques that they prefer.

2. Communication autonomy: Each of the individual DBMSs is free to make its own decision as to what type of information it wants to provide to the other DBMSs or to the software that controls their global execution.

3. Execution autonomy: Each DBMS can execute the transactions that are submitted to it in any way that it wants to.

In the taxonomy that we consider in the book, we will use a classification that covers the important aspects of these features. One alternative is tight integration where a single-image of the entire database is available to any user who wants to share the information, which may reside in multiple databases. From the users' perspective, the data is logically centralized in one database. In these tightly integrated systems, the data managers are implemented so that one of them is in control of the processing of each user request even if that request is serviced by more than one data manager. The data managers do not typically operate as independent DBMSs even though they usually have the functionality to do so.

Next we identify semiautonomous systems that consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable. Each of these DBMSs determines what parts of their own database they will make accessible to users of other DBMSs. They are not fully autonomous systems because they need to be modified to enable them to exchange information with one another.

The last alternative that we consider is total isolation where the individual systems are stand-alone DBMSs, which know neither of the existence of other DBMSs nor how to communicate with them. In such systems, the processing of user transactions

that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.

Whereas autonomy refers to the distribution of control, the distribution dimension of the taxonomy deals with data. We consider two cases, namely, either the data is physically distributed over multiple sites that communicate with each other over some form of communication medium or it is stored at only one site.

Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers. The important ones from the perspective of this book relate to data models, query languages, and transaction management protocols. Representing data with different modeling tools creates heterogeneity because of the inherent expressive powers and limitations of individual data models. Heterogeneity in query languages not only involves the use of completely different data access paradigms in different data models (set-at-a-time access in relational systems versus record-at-a-time access in network and hierarchical systems), but also covers differences in languages even when the individual systems use the same data model. Different query languages that use the same data model often select very different methods for expressing identical requests (e.g., DB2 uses SQL, while INGRES uses QUEL).6

Let us consider the architectural alternatives starting at the origin in Figure 3.4 and moving along the autonomy dimension. The first classes of systems are those which are logically integrated. Such systems can be given the generic name composite systems. If there is no distribution or heterogeneity, the system is a set of multiple DBMSs that are logically integrated. There are not many examples of such systems, but they may be suitable for shared-everything multiprocessor systems. If heterogeneity is introduced, one has multiple data managers that are heterogeneous but provide an integrated view to the user. In the past, some work was done in this class where systems were designed to provide integrated access to network, hierarchical, and relational databases residing on a single machine. The more interesting case is where the database is distributed even though an integrated view of the data is provided to users.

Next in the autonomy dimension are semiautonomous systems, which are commonly, termed federated DBMS. As specified before, the component systems in a federated environment have significant autonomy in their execution, but their participation in a federation indicate that they are willing to cooperate with others in executing user requests that access multiple databases. Similar to logically integrated

systems discussed above, federated systems can be distributed or single-site, homogeneous or heterogeneous.

If we move to full autonomy, we get what we call the class of multi database system (MDBS) architectures. Without heterogeneity or distribution, an MDBS is an interconnected collection of autonomous databases. A multi database management system (multi-DBMS) is the software that provides for the management of this collection of autonomous databases and transparent access to it. If the individual databases that make up the MDBS are distributed over a number of sites, we have a distributed MDBS. The organization of a distributed MDBS as well as its management is quite different from that of a distributed DBMS. We discuss this issue in more detail in the upcoming sections. At this point it suffices to point out that the fundamental difference is one of the levels of autonomy of the local data managers. Centralized or distributed multi database systems can be homogeneous or heterogeneous.

The fundamental point of the foregoing discussion is that the distribution of databases, their possible heterogeneity, and their autonomy are orthogonal issues. Since our concern in this book is on distributed systems, it is more important to note the orthogonal between autonomy and heterogeneity. Thus it is possible to have autonomous distributed databases that are not heterogeneous. In that sense, the more important issue is the autonomy of the databases rather than their heterogeneity. In other words, if the issues related to the design of a distributed multi database are resolved, introducing heterogeneity may not involve significant additional difficulty. This, of course, is true only from the perspective of database management; there may still be significant heterogeneity problems from the perspective of the operating system and the underlying hardware.

It is fair to claim that the fundamental issues related to multi database systems can be investigated without reference to their distribution. The additional considerations that distribution brings, in this case, are no different from those of logically integrated distributed database systems. Therefore, in this chapter we consider architectural models of logically integrated distributed DBMSs and multi database systems.

3.4.1 Distributed DBMS Architecture

Let us start the description of the architecture by looking at the data organizational view. We first note that the physical data organization on each machine may be, and probably is, different. This means that there needs to be an individual

internal schema definition at each site, which we call the local internal schema (LIS). The enterprise view of the data is described by the global conceptual schema (GCS), which is global because it describes the logical structure of the data at all the sites.

This architecture model, depicted in Figure 3.5, provides the levels of transparency discussed. Data independence is supported since the model is an extension of ANSI/SPARC, which provides such independence naturally. Location and replication transparencies are supported by the definition of the local and global conceptual schemas and the mapping in between. Network transparency, on the other hand, is supported by the definition of the global conceptual schema. The user queries data irrespective of its location or of which local component of the distributed database system will service it. As mentioned before, the distributed DBMS translates global queries into a group of local queries, which are executed by distributed DBMS components at different sites that communicate with one another.

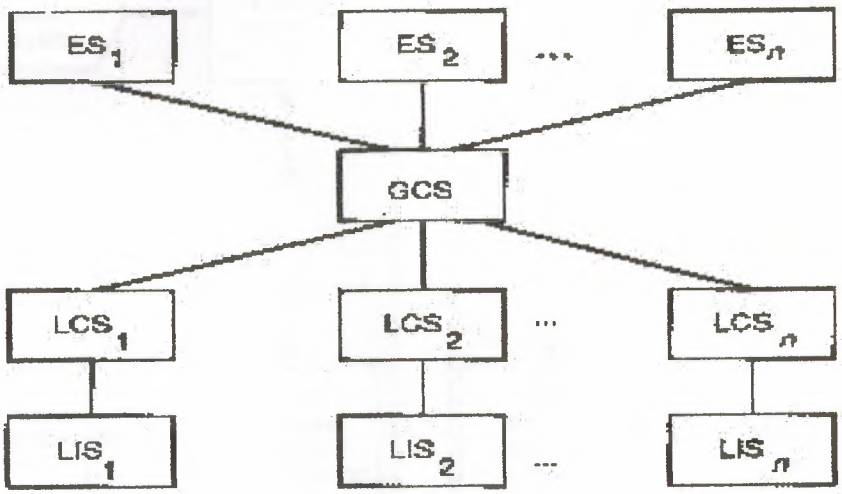

Figure 3.5: Distributed Database Reference Architecture

One component handles the interaction with users, and another deals with the storage. The first major component, which we call the user processor, consists of four elements:

1. The user interface handler is responsible for interpreting user commands as they come in, and formatting the result data as it is sent to the user.

2. The semantic data controller uses the integrity constraints and authorizations that are defined as part of the global conceptual schema to check if the user query can be processed.

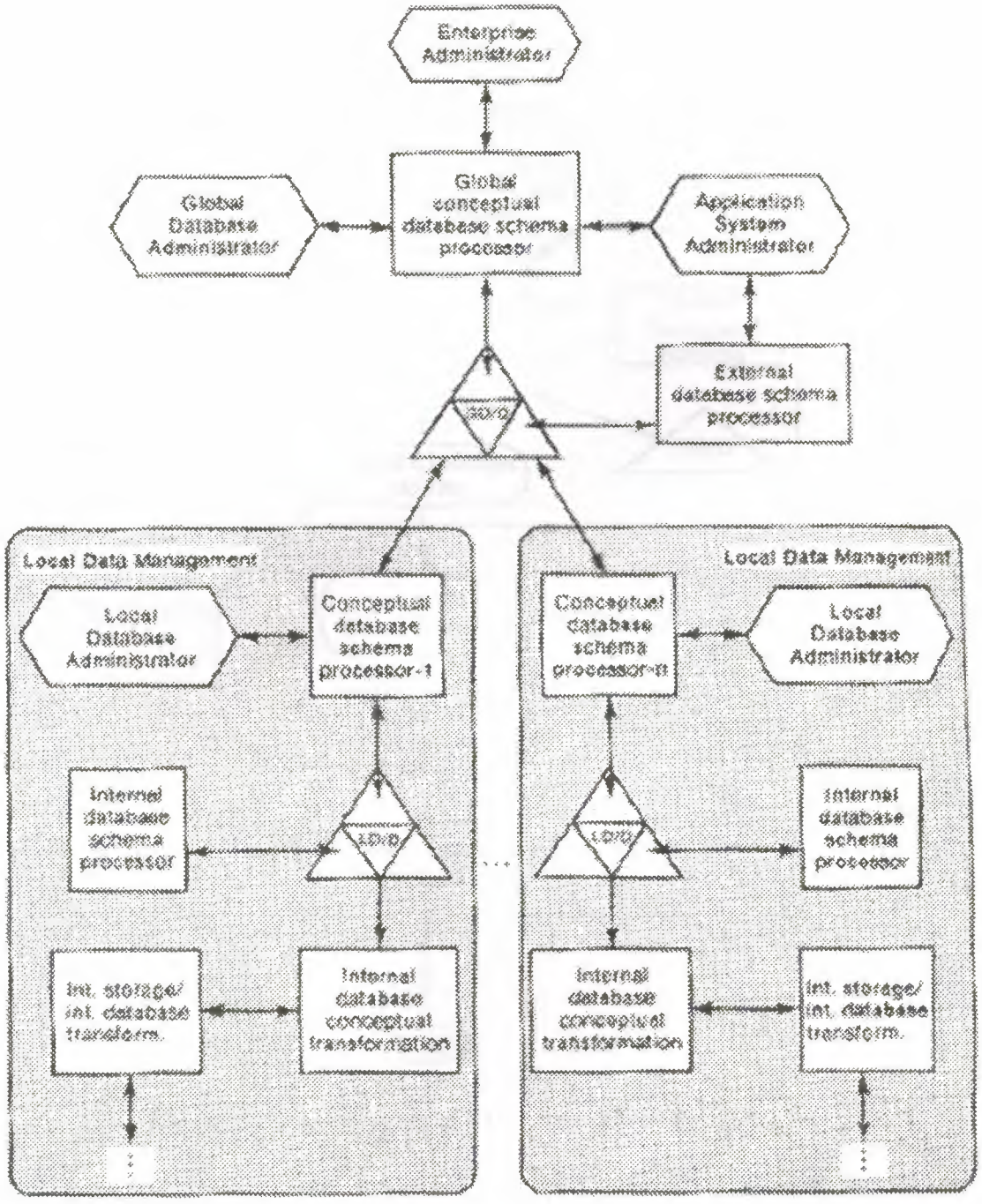

Figure 3.6: Functional Schematic of an Integrated Distributed DBMS

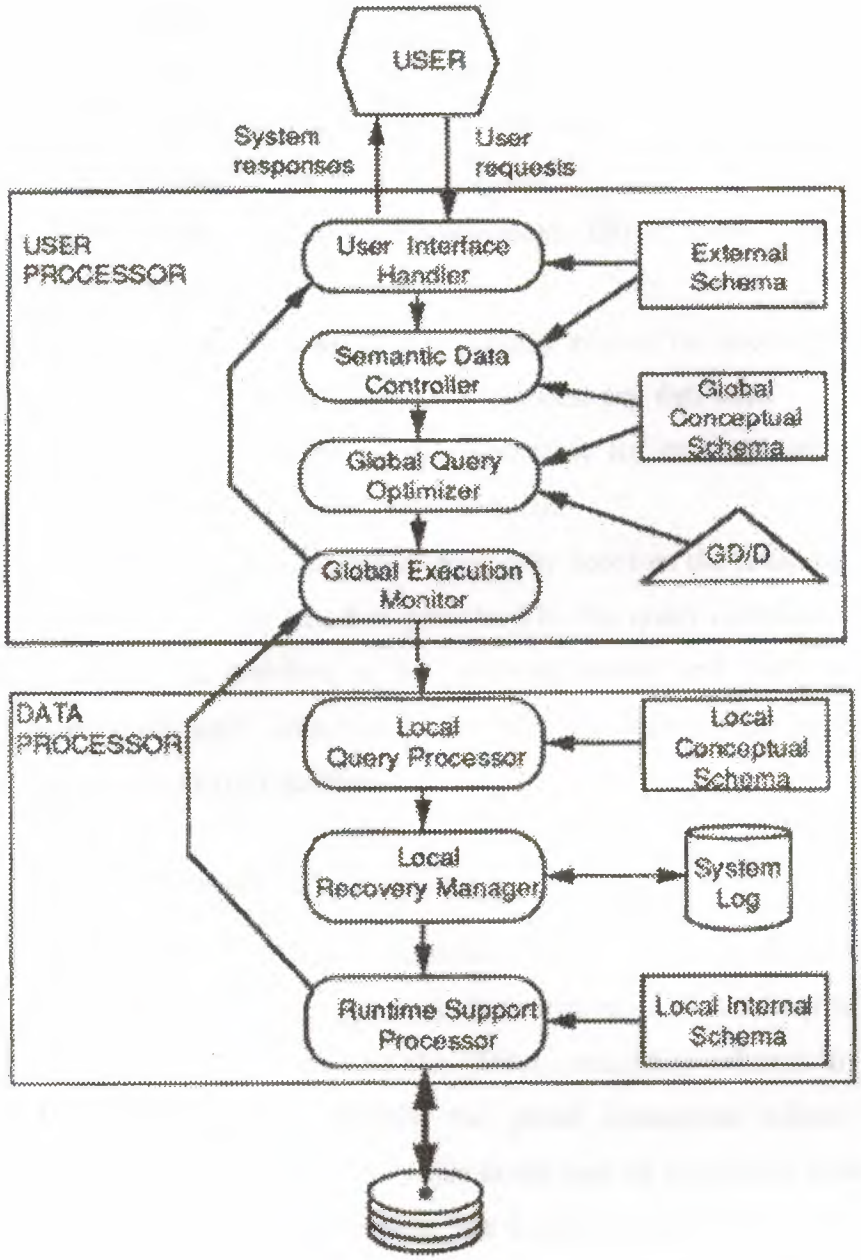

Figure 3.7: Components of a Distributed DBMS

3. The global query optimizer and decomposer determine an execution strategy to minimize a cost function, and translate the global queries into local ones using the global and local conceptual schemas as well as the global directory/dictionary. The global query optimizer is responsible, among other things, for generating the best strategy to execute distributed join operations.

4. The distributed execution monitor coordinates the distributed execution of the user request. The execution monitor is also called the distributed transaction manager. In executing queries in a distributed fashion, the execution monitors at various sites may, and usually do, communicate with one another.

The second major component of a distributed DBMS is the data processor and consists of three elements:

1. The local query optimizer, which actually acts as the access path selector, is responsible for choosing the best access path7 to access any data item.

2. The local recovery manager is responsible for making sure that the local database remains consistent even when failures occur.

3. The run-time support processor physically accesses the database according to the physical commands in the schedule generated by the query optimizer. The run-time support processor is the interface to the operating system and contains the database buffer (or cache) manager, which is responsible for maintaining the main memory buffers and managing the data accesses.

3.4.2 MDBS Architecture

The differences in the level of autonomy between the distributed multi DBMSs and distributed DBMSs are also reflected in their architectural models. The fundamental difference relates to the definition of the global conceptual schema. In the case of logically integrated distributed DBMSs, the global conceptual schema defines the conceptual view of the entire database, while in the case of distributed multi-DBMSs, it represents only the collection of some of the local databases that each local DBMS wants to share. Thus the definition of a global database is different in MDBSs than in distributed DBMSs. In the latter, the global database is equal to the union of local databases, whereas in the former it is only a subset of the same union. There are even arguments as to whether the global conceptual schema should even exist in multi database systems. This question forms the basis of our architectural discussions in this section.

Models using a global conceptual schema: In a MDBS, the GCS is defined by integrating either the external schemas of local autonomous databases or parts of their local conceptual schemas. Furthermore, users of a local DBMS define their own views on the local database and do not need to change their applications if they do not want to access data from another database. This is again an issue of autonomy.

Designing the global conceptual schema in multi database systems involves the integration of either the local global conceptual schemas or the local external schemas. A major difference between the design of the GCS in multi-DBMSs and in logically integrated distributed DBMSs is that in the former the mapping is from local conceptual schemas to a global schema. In the latter, however, mapping is in the reverse

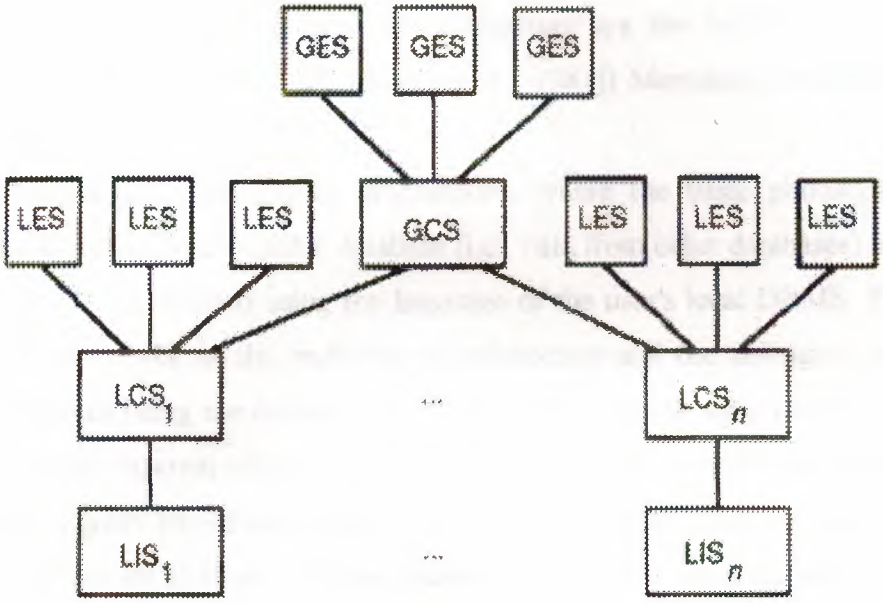

Figure 3.8: MDBS Architecture with a GCS

Direction: This is because the design in the former is usually a bottom-up process, whereas in the latter it is usually a top-down procedure. Further more, if heterogeneity exists in the multi database system, a canonical data model has to be found to define the GCS.

Once the GCS has been designed, views over the global schema can be defined for users who require global access. It is not necessary for the GES and GCS to be defined using the same data model and language; whether they do or not determines whether the system is homogeneous or heterogeneous.

If heterogeneity exists in the system, then two implementation alternatives exist: anilingual and multilingual. An anilingual multi-DBMS requires the users to utilize possibly different data models and languages when both a local database and the global database are accessed. The identifying characteristic of anilingual systems is that any application that accesses data from multiple databases must do so by means of an external view that is defined on the global conceptual schema. This means that the user of the global database is effectively a different user than those who access only a local

database, utilizing a different data model and a different data language. Thus, one application may have a local external schema (LES) defined on the local conceptual schema as well as a global external schema (GES) defined on the global conceptual schema. The different external view definitions may require the use of different access languages. Figure 3.8 actually depicts the data logical model of a anilingual database system that integrates the local conceptual schemas (or parts of them) into a global conceptual schema. Examples of such an architecture are the MULTIBASE system ([Landers and Rosenberg, 1982] and [Smith et al., 1981]) Mermaid [Templeton et al., 1987] and DDTS.

An alternative is multilingual architecture, where the basic philosophy is to permit each user to access the global database (i.e., data from other databases) by means of an external schema, defined using the language of the user's local DBMS. The GCS definition is quite similar in the multilingual architecture and the anilingual approach, the major difference being the definition of the external schemas, which are described in the language of the external schemas of the local database. Assuming that the definition is purely local, a query issued according to a particular schema is handled exactly as any query in the centralized DBMSs. Queries against the global database are made using the language of the local DBMS, but they generally require some processing to be mapped to the global conceptual schema.

The multilingual approach obviously makes querying the databases easier from the user's perspective. However, it is more complicated because we must deal translation of queries at run time. The multilingual approach is used in Sirius-Delta and in the HD-DBMS project.

Models without a global conceptual schema: The existence of a global conceptual schema in a multi database system is a controversial issue. There are re-B searchers who even define a multi database management system as one that manages several databases without a global schema. It is argued that the absence of a GCS is a significant advantage of multi database systems over distributed database systems. One prototype system that has used this architectural model is the MRDSM project. **Identifies two layers:** The local system layer and the multi database layer on top of it. The local system layer consists of a number of DBMSs, which present to the multi database layer the part of their local database they are willing to share with users of other databases. This shared data is presented either as the actual local conceptual

schema or as a local external schema definition. If heterogeneity is involved, each of these schemas, LCSi, may use a different data model.

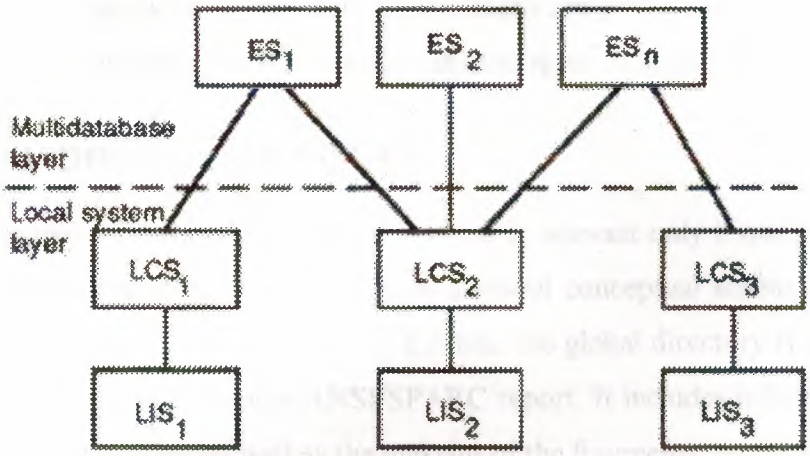

Figure 3.9: MDBS Architecture without a GCS

Above this layer, external views are constructed where each view may be defined on one local conceptual schema or on multiple conceptual schemas. Thus the responsibility of providing access to multiple (and maybe heterogeneous) databases is delegated to the mapping between the external schemas and the local conceptual schemas. This is fundamentally different from architectural models that use a global conceptual schema, where this responsibility is taken over by the mapping between the global conceptual schema and the local ones. This shift in responsibility has a practical consequence. Access to multiple databases is provided by means of a powerful language in which user applications are written.

Federated database architectures, which we discussed briefly, do not use a global conceptual schema either. In the specific system described in, each local DBMS defines an export schema, which describes the data it is willing to share with others. In the terminology that we have been using, the global database is the union of all the export schemas.

The component-based architectural model of a multi-DBMS is significantly different from a distributed DBMS. The fundamental difference is the existence of full fledged DBMSs, each of which manages a different database. The MDBS provides a layer of software that runs on top of these individual DBMSs and provides users with the facilities of accessing various databases. Depending on the existence (or lack) of the global conceptual schema or the existence of heterogeneity (or lack of it), the contents

of this layer of software would change significantly. Note that Figure 3.10 represents a non distributed multi-DBMS. If the system is distributed, we would need to replicate the multi database layer to each site where there is a local DBMS that participates in the system. Also note that as far as the individual DBMSs are concerned, the MDBS layer is simply another application that submits requests and receives answers.

3.5 GLOBAL DIRECTORY ISSUES

The discussion of the global directory issues is relevant only if one talks about a distributed DBMS or a multi-DBMS that uses a global conceptual schema. Otherwise, there is no concept of a global directory. If it exists, the global directory is an extension of the dictionary as described in the ANSI/SPARC report. It includes information about the location of the fragments as well as the makeup of the fragments.

As stated earlier, the directory is itself a database that contains meta-data about the actual data stored in the database. Therefore, the techniques with respect to distributed database design also apply to directory management. Briefly, a directory may be either global to the entire database or local to each site. In other words, there might be a single directory containing information about all the data in the database, or a number of directories, each containing the information stored at one site. In the latter case, we might either build hierarchies of directories to facilitate searches, or implement a distributed search strategy that involves considerable communication among the sites holding the directories.

The second issue has to do with location. The directory may be maintained centrally at one site or in a distributed fashion by distributing it over a number of sites.

Keeping the directory at one site might increase the load at that site, thereby causing a bottleneck as well as increasing message traffic around that site. Distributing it over a number of sites, on the other hand, increases the complexity of managing directories. In the case of multi-DBMSs, the choice is dependent on whether or not the system is distributed. If it is, the directory is always distributed; otherwise of course, it is maintained centrally.

The final issue is replication. There may be a single copy of the directory or multiple copies. Multiple copies would provide more reliability, since the probability of reaching one copy of the directory would be higher. Furthermore, the delays in accessing the directory would be lower, due to less contention and the relative

proximity / of the directory copies. On the other hand, keeping the directory up to date would be considerably more difficult, since multiple copies would need to be updated.

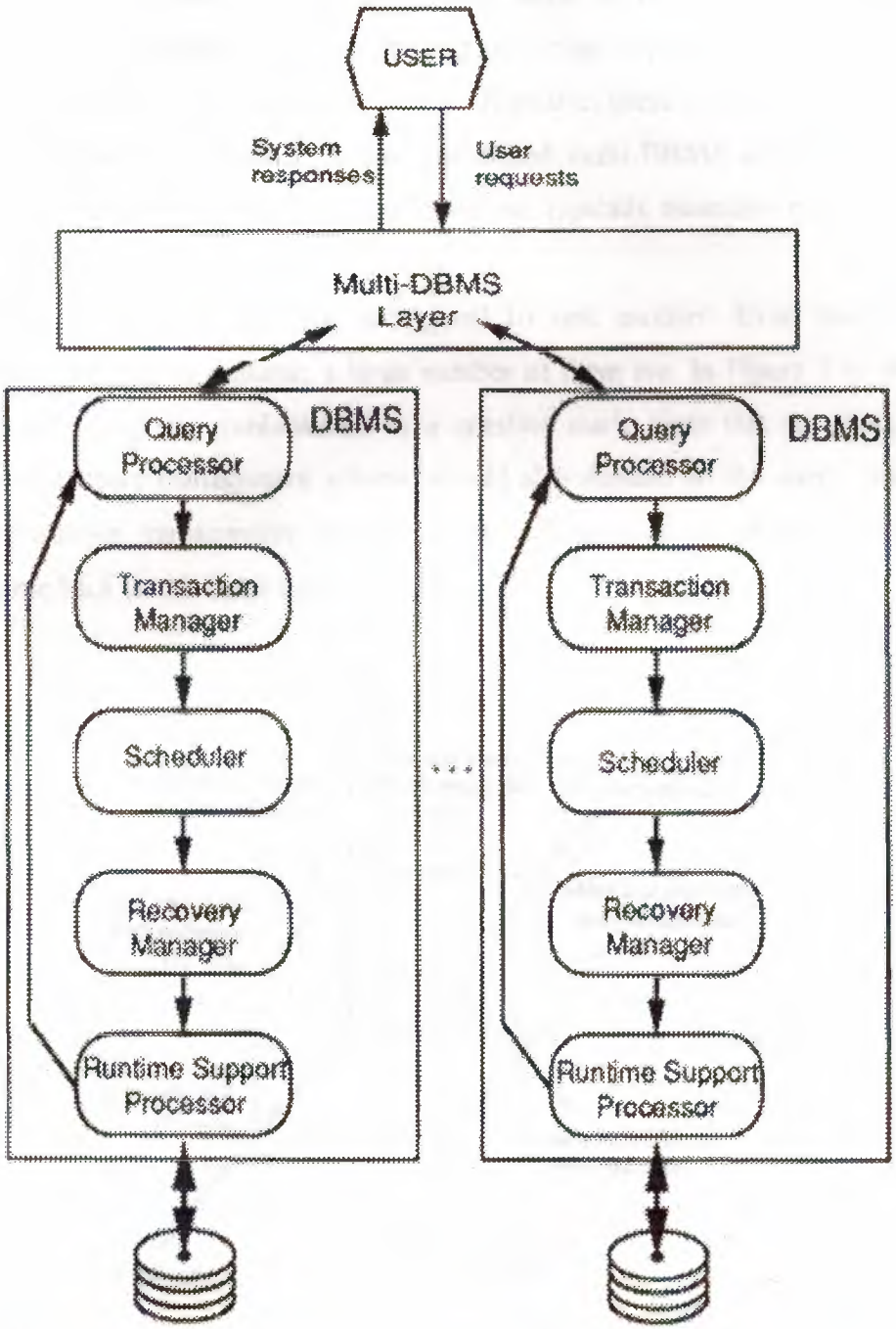

Figure 3.10: Components of an MDBS

Therefore, the choice should depend on the environment in which the system operates and should be made by balancing such factors as the response-time requirements, the size of the directory, the machine capacities at the sites, the reliability requirements, and the volatility of the directory (i.e., the amount of change experienced by the database, which would cause a change to the directory). Of course, these choices are valid only in the case of a distributed DBMS. A non distributed multi-DBMS always maintains a single copy of the directory, while a distributed one typically maintains multiple copies, one at each site.

These three dimensions are orthogonal to one another. Even though some combinations may not be realistic, a large number of them are. In Figure 3.11 we have designated the unrealistic combinations by a question mark. Note that the choice of an appropriate directory management scheme should also depend on the query processing and the transaction management techniques that will be used in subsequent chapters. We will come back to this issue again.

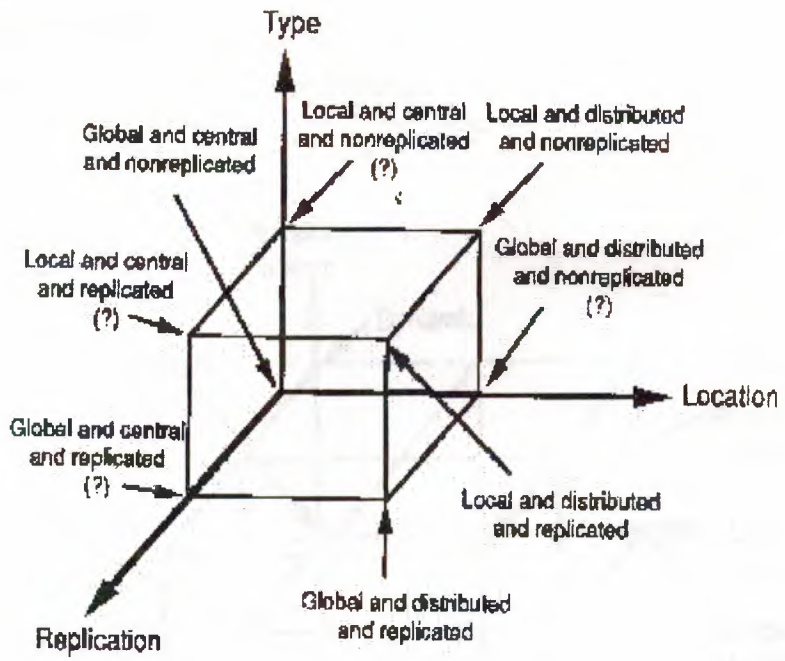

Figure 3.11: Alternative Directory Management Strategies

CHAPTER FOUR

DISTRIBUTED DATABASE DESIGN

The design of a distributed computer system involves making decisions on the placement of data and programs across the sites of a computer network, as well as possibly designing the network itself. In the case of distributed DBMSs, the distribution of applications involves two things: the distribution of the distributed DBMS software and the distribution of the application programs that run on it. The former is not a significant problem, since we assume that a copy of the distributed DBMS software exists at each site where data is stored. In this chapter we do not concern ourselves with application program placement either. Furthermore, we assume that the network has already been designed, or will be designed at a later stage, according to the decisions related to the distributed database design. We concentrate on distribution of data. It has been suggested that the organization of distributed systems can be investigated along three orthogonal dimensions [Levin and Morgan, 1975]:

1. Level of sharing

2. Behavior of access patterns

3. Level of knowledge on access pattern behavior

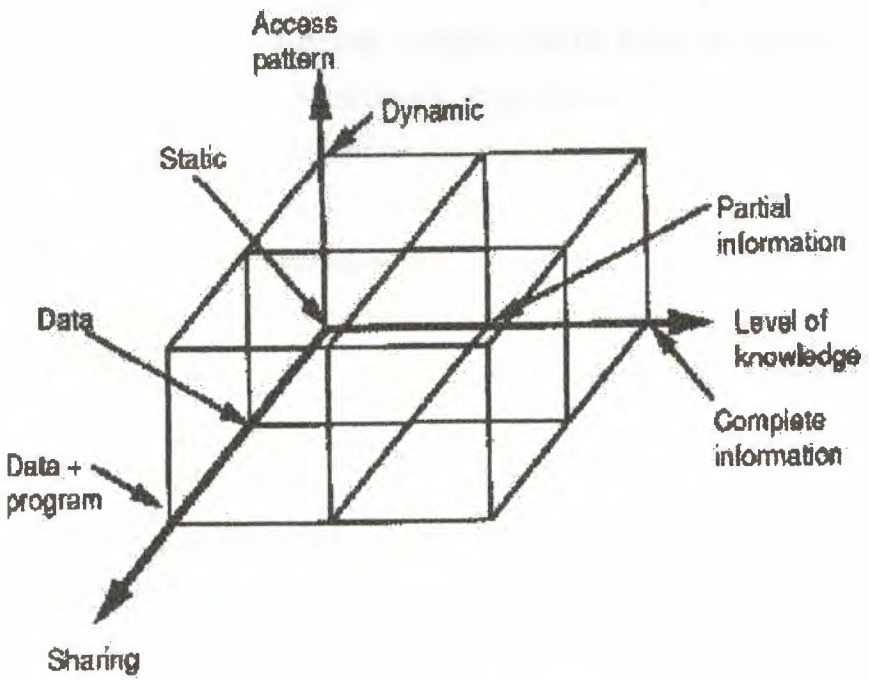

Figure 4.1: Framework of Distribution

Figure 4.1 depicts the alternatives along these dimensions. In terms of the level of sharing, there are possibilities. First, there is no sharing: each application and its data execute at one site, and there is no communication with any other program or access to any data file at other sites. This characterizes the very early days of networking and is probably not very common today. We then find the level of data sharing; all the programs are replicated at all the sites, but data files are not. Accordingly, user requests are handled at the site where they originate and the necessary data files are moved around the network. Finally, in data-plus-program sharing, both data and programs may be shared, meaning that a program at a given site can request a service from another program at a second site, which, in turn, may have to access a data file located at a third site.

Levin and Morgan draw a distinction between data sharing and data-plus-program sharing to illustrate the differences between homogeneous and heterogeneous distributed computer systems. They indicate, correctly, that in a heterogeneous environment it is usually very difficult, if not impossible, to execute a given program on different hardware under a different operating system. It might, however, be possible to move data around relatively easily.

Along the second dimension of access pattern behavior, it is possible to identify two alternatives. The access patterns of user requests may be static, so that they do not change over time, or dynamic. It is obviously considerably easier to plan for and manage the static environments than would be the case for dynamic distributed systems. Unfortunately, it is difficult to find many real-life distributed applications that would be classified as static. The significant question, then, is not whether a system is static or dynamic, but how dynamic it is. Incidentally, it is along this dimension that the relationship between the distributed database design and query processing is established.

The third dimension of classification is the level of knowledge about the access pattern behavior. One possibility, of course, is that the designers do not have any information about how users will access the database. This is a theoretical possibility, but it is very difficult, if not impossible, to design a distributed DBMS that can effectively cope with this situation. The more practical alternatives are that the

designers have complete information, where the access patterns can reasonably be predicted and do not deviate significantly from these predictions, and partial information, where there are deviations from the predictions.

The distributed database design problem should be considered within this general framework. In all the cases discussed, except in the no-sharing alternative, new problems are introduced in the distributed environments which are not relevant in a centralized setting. In this chapter it is our objective to focus on these unique problems. The outline of this chapter is as follows. In Section 4.1 we discuss briefly two approaches to distributed database design: the top-down and the bottom-up design strategies. The details of the top-down approach are given in Sections 4.3 and 4.4, while the details of the bottom-up approach are postponed to another chapter. Prior to the discussion of these alternatives, in Section 4.2 we present the issues in distribution design.

4.1 ALTERNATIVE DESIGN STRATEGIES

Two major strategies that have been identified [Ceri et al., 1987] for designing distributed databases are the top-down approach and the bottom-up approach. As the names indicate, they constitute very different approaches to the design process. But as any software designer knows, real applications are rarely simple enough to fit nicely in either of these alternatives. It is therefore important to keep in mind that in most database designs, the two approaches may need to be applied to complement one another.

4.1.1 Top-Down Design Process

A framework for this process is shown in Figure 4.2. The activity begins with a requirements analysis that defines the environment of the system and "elicits both the data and processing needs of all potential database users" [Yao et al., 1982a]. The requirements study also specifies where the final system is expected to stand with respect to the objectives of a distributed DBMS as identified in Section 1.3. Tb reiterates, these objectives are defined with respect to performance, reliability and availability, economics, and expandability (flexibility).

The requirements document is input to two parallel activities: view design and conceptual design. The view design activity deals with defining the interfaces for end users. The conceptual design, on the other hand, is the process by which the enterprise is examined to determine entity types and relationships among these entities. One can possibly divide this process into two related activity groups [Davenport, 1981]: entity analysis and functional analysis. Entity analysis is concerned with determining the entities, their attributes, and the relationships among them. Functional analysis, on the other hand, is concerned with determining the fundamental functions with which the modeled enterprise is involved. The results of these two steps need to be cross-referenced to get a better understanding of which functions deal with which entities.

There is a relationship between the conceptual design and the view design. In one sense, the conceptual design can be interpreted as being an integration of user views. Even though this view integration activity is very important, the conceptual model should support not only the existing applications, but also future applications. View integration should be used to ensure that entity and relationship requirements for all the views are covered in the conceptual schema.

In conceptual design and view design activities the user needs to specify the data entities and must determine the applications that will run on the database as well as statistical information about these applications. Statistical information includes the specification of the frequency of user applications, the volume of various information , and the like. Note that from the conceptual design step come the definition of global conceptual schema discussed in Section 4.3. We have not yet considered the implications of the distributed environment; in fact, up to this point, the process is identical to that in a centralized database design.

The global conceptual schema (GCS) and access pattern information collected as a result of view design are inputs to the distribution design step. The objective at this stage, which is the focus of this chapter, is to design the local conceptual schemas (LCSs) by distributing the entities over the sites of the distributed system. It is possible, of course, to treat each entity as a unit of distribution. Given that we use the relational model as the basis of discussion in this book, the entities correspond to relations.

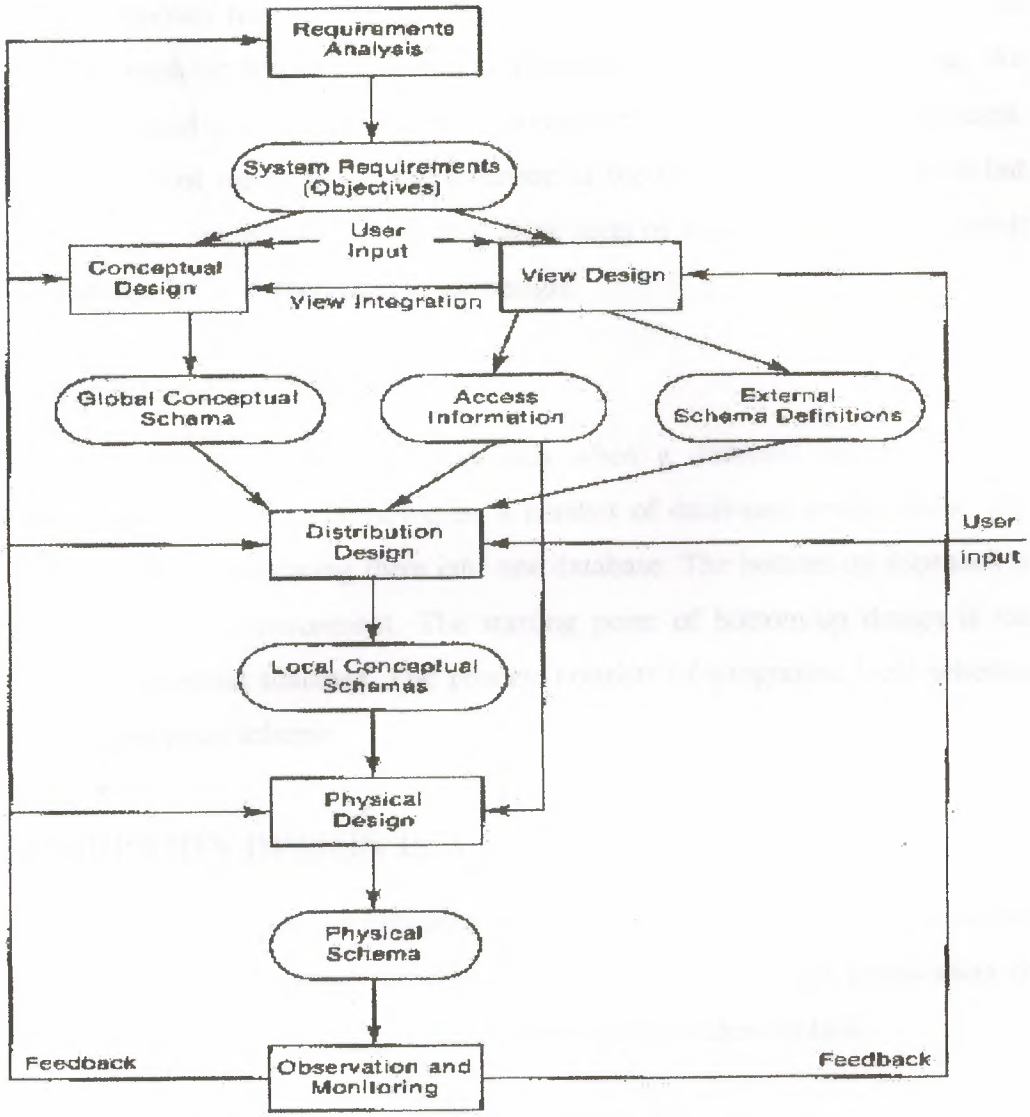

Figure 4.2: Top-Down Design process

Rather than distributing relations, it is quite common to divide them into sub relations, called fragments, which are then distributed. Thus the distribution design activity consists of two steps: fragmentation and allocation. These are the major issues that are treated in this chapter, so we delay discussing them until later sections.

The last step in the design process is the physical design, which maps the local conceptual schemas to the physical storage devices available at the corresponding sites. The inputs to this process are the local conceptual schema and access pattern information about the fragments in these.

It is well known that the design and development activity of any kind is an ongoing process requiring constant monitoring and periodic adjustment and tuning. We have therefore included observation and monitoring as a major activity in this process. Note that one does not monitor only the behavior of the database implementation but also the suitability of user views. The result is some form of feedback, which may result in backing up to one of the earlier steps in the design.

4.1.2 Bottom-Up Design Process

Top-down design is a suitable approach when a database system is being designed from scratch. Commonly, however, a number of databases already exist, and the design task involves integrating them into one database. The bottom-up approach is suitable for this type of environment. The starting point of bottom-up design is the individual local conceptual schemas. The process consists of integrating local schemas into the global conceptual schema.

4.2 DISTRIBUTION DESIGN ISSUES

In the preceding section we indicated that the relations in a database schema are usually decomposed into smaller fragments, but we did not offer any justification or details for this process. The objective of this section is to fill in these details.

The following set of interrelated questions covers the entire issue. We will there fore seek to answer them in the remainder of this section.

- Why fragment at all?
- How should we fragment?
- How much should we fragment?
- Is there any way to test the correctness of decomposition?
- How should we allocate?
- What is the necessary information for fragmentation and allocation?

4.2.1 Reasons for Fragmentation

From a data distribution viewpoint, there is really no reason to fragment data. After all, in distributed file systems, the distribution is performed on the basis of

enfolds. In fact, the earlier work dealt specifically with the allocation of files to nodes on a computer network.

With respect to fragmentation, the important issue is the appropriate unit of distribution. A relation is not a suitable unit, for a number of reasons. First, application views are usually subsets of relations. Therefore, the locality of accesses of applications is defined not on entire relations but on their subsets. Hence it is only natural to consider subsets of relations as distribution units.

Second, if the applications that have views defined on a given relation reside at different sites, two alternatives can be followed, with the entire relation being the unit of distribution. Either the relation is not replicated and is stored at only one site, or it is replicated at all or some of the sites where the applications reside. The former results in an unnecessarily high volume of remote data accesses . The latter, on the other hand, has unnecessary replication, which causes problems in executing updates (to be discussed later) and may not be desirable if storage is limited.

Finally, 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 sub queries that operate on fragments. Thus fragmentation typically increases the level of concurrency and therefore the system throughput.

For the sake of completeness, we should also indicate the 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. 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 second problem is related to semantic data control, specifically to integrity checking. As a result of fragmentation, attributes participating in a dependency may be decomposed into different fragments which might be allocated to different sites. In this

case, even the simpler task of checking for dependencies would result in chasing after data in a number of sites

4.2.2 Fragmentation Alternatives

Relation instances are essentially tables, so the issue is one of finding alternative ways of dividing a table into smaller ones. There are clearly two alternatives for this: dividing it horizontally or dividing it vertically.

Example 4.1

Figure 4.5 shows the J relation of Figure 4.3 partitioned vertically into two sub relations, J_1 and J_2. J_1 contains only the information about project budgets, whereas J_2 contains project names and locations. It is important to notice that the key to the relation (JNO) is included in both fragments.

The fragmentation may, of course, be nested. If the nestlings are of different types, one gets hybrid fragmentation. Even though we do not treat hybrid fragmentation as a primitive type of fragmentation strategies, it is quite obvious that many real-life partitioning may be hybrid.

E

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

G

ENO	JNO	RESP	DUR
E1	J1	Manager	12
E2	J1	Analyst	24
E2	J2	Analyst	6
E3	J3	Consultant	10
E3	J4	Engineer	48
E4	J2	Programmer	18
E5	J2	Manager	24
E6	J4	Manager	48
E7	J3	Engineer	36
E8	J3	Manager	40

J

JNO	JNAME	BUDGET	LOC
J1	Instrumentation	150000	Montreal
J2	Database Develop.	135000	New York
J3	CAD/CAM	250000	New York
J4	Maintenance	310000	Paris

S

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

Figure 4.3: Modified Example Database

J_1

JNO	JNAME	BUDGET	LOC
J1	Instrumentation	150000	Montreal
J2	Database Develop.	135000	New York

J_2

JNO	JNAME	BUDGET	LOC
J3	CAD/CAM	255000	New York
J4	Maintenance	310000	Paris

Figure 4.4: Example of Horizontal Partitioning

J_1

JNO	BUDGET
J1	150000
J2	135000
J3	250000
J4	310000

J_2

JNO	JNAME	LOC
J1	Instrumentation	Montreal
J2	Database Develop.	New York
J3	CAD/CAM	New York
J4	Maintenance	Paris

Figure 4.5: Example of Vertical Partitioning

4.2.3 Degree of Fragmentation

The extent to which the database should be fragmented is an important decision that affects the performance of query execution. In fact, the issues in Section 4.2.1 concerning the reasons for fragmentation constitute a subset of the answers to the question we are addressing here. 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 topples (in the case of horizontal fragmentation) or to the level of individual attributes (in the case of vertical fragmentation).

We have already addressed the adverse effects of very large and very small units of fragmentation. 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. The issue is, how? In general,

the applications need to be characterized with respect to a number of parameters. According to the values of these parameters, individual fragments can be identified.

4.2.4 Correctness Rules of Fragmentation

It is important to note the similarity between the fragmentation of data for distribution (specifically, vertical fragmentation) and the normalization of relations. Thus fragmentation rules similar to the normalization principles can be defined.

We will enforce the following three rules during fragmentation, which, together, ensure that the database does not undergo semantic change during fragmentation.

1. Completeness: If a relation instance R is decomposed into fragments R1, R_2,..., R_n, each data item that can be found in R can also be found in one or more of R_i's. This property, which is identical to the lossless decomposition property of normalization, is also important in fragmentation since it ensures that the data in a global relation is mapped into fragments without any loss [Grant, 1984]. Note that in the case of horizontal fragmentation, the "item" typically refers to a topple, while in the case of vertical fragmentation, it refers to an attribute.

2. Reconstruction: If a relation R is decomposed into fragments R_1, R_2,..., R_n, it should be possible to define a relational operator y such that

$$R = \nabla R_i, \quad \forall R_i \in F_R$$

The operator ∇ will be different for the different forms of fragmentation; it is important, however, that it can be identified. The reconstruct ability of the relation from its fragments ensures that constraints defined on the data in the form of dependencies are preserved.

3. Disjoint ness: If a relation R is horizontally decomposed into fragments R_1, R_2, ..., R_n and data item d_i is in R_j, it is not in any other fragment R_k ($k \neq j$). This criterion ensures that the horizontal fragments are disjoint. If relation R is vertically decomposed, its primary key attributes are typically repeated in all its fragments. Therefore, in case of vertical partitioning, disjoint ness is defined only on the no primary key attributes of a relation.

4.2.5 Allocation Alternatives

Assuming that the database is fragmented properly, one has to decide on the allocation of the fragments to various sites on the network. When data is allocated, it may either be replicated or maintained as a single copy. The reasons for replication are reliability and efficiency of read-only queries. If there are multiple copies of a data item, there is a good chance that some copy of the data will be accessible somewhere even when system failures occur. Furthermore, read-only queries that access the same data items can be executed in parallel since copies exist on multiple sites. On the other hand, the execution of update queries cause trouble since the system has to ensure that all the copies of the data are updated properly. Hence the decision regarding replication is a trade-off which depends on the ratio of the read-only queries to the update queries. This decision affects almost all of the distributed DBMS algorithms and control functions.

A no replicated database (commonly called a partitioned database) contains 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). In the latter the number of copies of a fragment may be an input to the allocation algorithm or a decision variable whose value is determined by the algorithm. Figure 4.6 compares these three replication alternatives with respect to various distributed DBMS functions.

	Full replication	Partial replication	Partitioning
QUERY PROCESSING	Easy	Same difficulty	
DIRECTORY MANAGEMENT	Easy or nonexistent	Same difficulty	
CONCURRENCY CONTROL	Moderate	Difficult	Easy
RELIABILITY	Very high	High	Low
REALITY	Possible application	Realistic	Possible application

Figure 4.6: Comparison of Replication Alternatives

4.2.6 Information Requirements

One aspect of distribution design is that too many factors contribute to an optimal design. The logical organization of the database, the location of the applications, the access characteristics of the applications to the database, and the properties of the computer systems at each site all have an influence on distribution decisions. This makes it very complicated to formulate a distribution problem.

The information needed for distribution design can be divided into four categories: database information, application information, communication network information, and computer system information. The latter two categories are completely quantitative in nature and are used in allocation models rather than in fragmentation algorithms. We do not consider them in detail here. Instead, the detailed information requirements of the fragmentation and allocation algorithms are discussed in their respective sections.

4.3 FRAGMENTATION

In this section we present the various fragmentation strategies and algorithms. As mentioned previously, there are two fundamental fragmentation strategies: horizontal and vertical. Furthermore, there is a possibility of nesting fragments in a hybrid fashion.

4.3.1 Horizontal Fragmentation

As we explained earlier, horizontal fragmentation partitions a relation along its topples. Thus each fragment has a subset of the topples of the relation. There are two versions of horizontal partitioning: primary and derived. Primary horizontal fragmentation of a relation is performed using predicates that are defined on that relation. Derived horizontal fragmentation, on the other hand, is the partitioning of a relation that result from predicates being defined on another relation.

Later in this section we consider an algorithm for performing both of these fragmentations. However, first we investigate the information needed to carry out horizontal fragmentation activity.

Information requirements of horizontal fragmentation

Database Information: The database information concerns the global conceptual schema. In this context it is important to note how the database relations are connected to one another, especially with joins. In the relational model, these relationships are also depicted as relations. However, in other data models, such as the entity-relationship (E-R) model [Chen, 1976], these relationships between database objects are depicted explicitly. In [Cheri et al., 1983] the relationship is also modeled explicitly, within the relational framework, for purposes of the distribution design. In the latter notation, directed links are drawn between relations that are related to each other by an equip-join operation.

Example 4.2

The links between database objects (i.e., relations in our case) should be quite familiar to those who have dealt with network models of data. In the relational model they are introduced as join graphs, which we discuss in detail in subsequent chapters on query processing. We introduce them here because they help to simplify the presentation of the distribution models we discuss later.

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 [Cheri et al., 1983]. More commonly used terms, within the relational framework, are source relation for owner and target relation for member. Let us define two functions: owner and member, both of which provide mappings from the set of links to the set of relations. Therefore, given a link, they return the member or owner relations of the link, respectively.

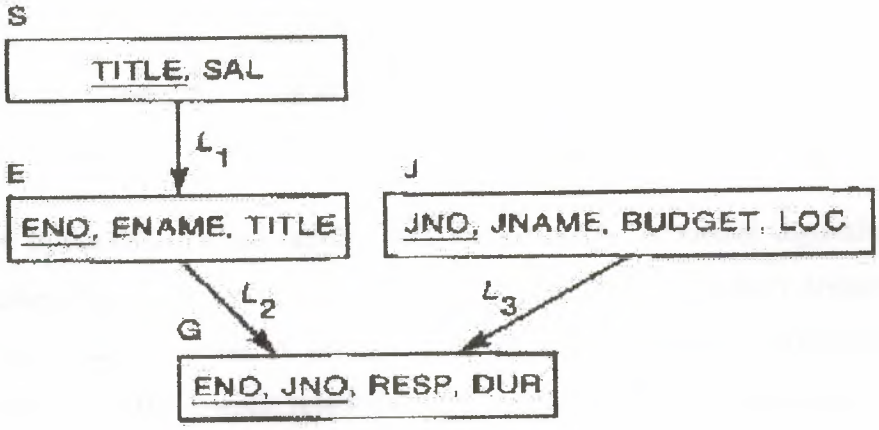

Figure 4.7: Expression of Relationships among Relations Using Links

Example 4.3

Given link L_1 of Figure 4.7, the owner and member functions have the following values:

$$\text{Owner } (L_1) = S$$

$$\text{Member } (L_1) = E$$

The quantitative information required about the database is the cardinality of each relation R, denoted card (R).

Application Information: As indicated previously in relation to Figure 4.2, both qualitative and quantitative information is required about applications. The qualitative information guides the fragmentation activity, whereas the quantitative information is incorporated primarily into the allocation models.

The fundamental qualitative information consists of the predicates used in user queries. If it is not possible to analyze all of the user applications to determine these predicates, one should at least investigate the most "important" ones. It has been suggested that as a rule of thumb, the most active 20% of user queries account for 80% of the total data accesses [Wielder-hold, 1982]. This "80/20 rule" may be used as a guideline in carrying out this analysis.

At this point we are interested in determining simple predicates. Given a relation $R(A_1,A_2, ..., A_n)$, where A{ is an attribute defined over domain Di, a simple predicate pj defined on R has the form

$$P_j : A_i \theta \text{ Value}$$

Where $\theta \in \{=, <, \neq, \leq, >, \geq\}$ and Value is chosen from the domain of Ai (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}.

Primary horizontal fragmentation: Before we present a formal algorithm for horizontal fragmentation, we should intuitively discuss the process for both primary and derived horizontal fragmentation. A primary horizontal fragmentation is defined by a selection operation on the owner relations of a database schema. Therefore, given relation R_i, its horizontal fragments are given by

$$R_i^j = \sigma_{Fi}(R_i),\ 1 \leq J \leq w$$

Where F_j is the selection formula used to obtain fragment R_i^j. Note that if F_j is in conjunctive normal form, it is a midterm predicate (m_{ij}). The algorithm we discuss will, in fact, insist that F_j be a midterm predicate.

Derived horizontal fragmentation: A derived horizontal fragmentation is defined on a member relation of a link according to a selection operation specified on its owner. It is important to remember two points. First, the link between the owner and the member relations is defined as an equip-join. Second, an equip-join can be implemented by means of semi joins. This second point is especially important for our purposes, since 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.

Accordingly, given a link L where owner (L) = S and member (L) = R, the derived horizontal fragments of R are defined as

$$R_i = R \triangleright\!< S_i, 1 \leq i \leq w$$

Where w is the maximum number of fragments that will be defined on R, and $S_i = \sigma F_i(S)$, where F_i is the formula according to which the primary horizontal

Fragment Si is defined.

Example 4.4

Consider link Li in Figure 4.7, where owner (Li) = S and member (Li) = E. Then we can group engineers into two groups according to their salary: those making less than or equal to $30,000, and those making more than $30,000. The two fragments Ei and E2 are defined as follows:

$$E_1 = E \triangleright\!< S_1$$
$$E_2 = E \triangleright\!< S_2$$

Where

$$S_1 = \sigma\, SAL \leq {}_{30000}(S)$$
$$S_2 = \sigma\, SAL > {}_{30000}(S)$$

The result of this fragmentation is depicted in Figure 4.8

E_1

ENO	ENAME	TITLE
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E7	R. Davis	Mech. Eng.

E_2

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Syst. Anal.
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E8	J. Jones	Syst. Anal.

Figure 4.8: Derived Horizontal Fragmentation of Relation E

To carry out a derived horizontal fragmentation, three inputs are needed: the set of partitions of the owner relation, the member relation, and the set of semi join predicates between the owner and the member. The fragmentation algorithm, then, is quite trivial, so we will not present it in any detail.

There is one potential complication that deserves some attention. In a database schema, it is common that there are more than two links into a relation R. In this case there is more than one possible derived horizontal fragmentation of R. The decision as to which candidate fragmentation to choose is based on two criteria:

1. The fragmentation with better join characteristics

2. The fragmentation used in more applications

Let us discuss the second criterion first. This is quite straightforward if we take into consideration the frequency with which applications access some data. If possible, one should try to facilitate the accesses of the "heavy" users so that their total impact on system performance is minimized.

Applying the first criterion, however, is not that straightforward. Consider, for example, the fragmentation we discussed. The effect of this fragmentation is that the join of the E and S relations to answer the

Query is assisted (1) by performing it on smaller relations (i.e., fragments), and (2) by potentially performing joins in a distributed fashion.

The first point is obvious. The fragments of E are smaller than E itself. Therefore, it will be faster to join any fragment of S with any fragment of E than to work with the relations themselves. The second point, however, is more important and is at the heart of distributed databases. If, besides executing a number of queries at different sites, we can execute one query in parallel, the response time or throughput of the system can be expected to improve. In the case of joins, this is possible under certain circumstances. There is only one link coming in or going out of a fragment. Such a join graph is called a simple graph. The advantage of a design where the join relationship between fragments is simple is that the member and owner of a link can be allocated to one site and the joins between different pairs of fragments can proceed independently and in parallel.

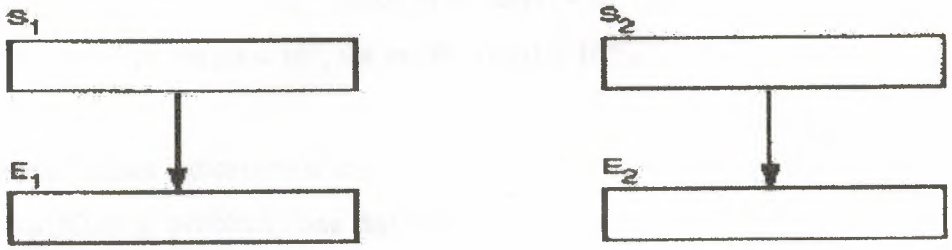

Figure 4.9: Join Graph between Fragments

Unfortunately, obtaining simple join graphs may not always be possible. In that case, the next desirable alternative is to have a design that results in a partitioned join graph. A partitioned graph consists of two or more sub graphs with no links between them. Fragments so obtained may not be distributed for parallel execution as easily as those obtained via simple join graphs, but the allocation is still possible.

4.3.2 Vertical Fragmentation

Remember that a vertical fragmentation of a relation R produces fragments R_1 ,R_2 ..., R_n , each of which contains a subset of R's attributes as well as the primary key of R. The objective of vertical fragmentation is to partition a relation into a set of smaller relations so that many of the user applications will run on only one fragment. In this context, an "optimal" fragmentation is one that produces a fragmentation scheme which minimizes the execution time of user applications that run on these fragments.

Vertical fragmentation has been investigated within the context of centralized database systems as well as distributed ones. Its motivation within the centralized context is as a design tool, which allows the user queries to deal with smaller relations,

thus causing a smaller number of pages. It has also been suggested that the most active sub relations can be identified and placed in a faster memory subsystem in those cases where memory hierarchies are supported.

Vertical partitioning is inherently more complicated than horizontal partitioning. This is due to the total number of alternatives that are available. For example, in horizontal partitioning, if the total number of simple predicates in P_r is n, there are 2^n possible midterm predicates that can be defined on it. In addition, we know that some of these will contradict the existing implications, further reducing the candidate fragments that need to be considered. In the case of vertical partitioning, however, if a relation has m non primary key attributes, the number of possible fragments is equal to B(m), which is the m^{th} Bell number. For large values of m, $B(m) \approx m^m$; for example, for m=10, B(m) $\approx$ 115,000, for m=15, $B(m) \approx 10^9$, for m=30, $B(m) = 10^{23}$.

These values indicate that it is futile to attempt to obtain optimal solutions to the vertical partitioning problem; one has to resort to heuristics. Two types of heuristic approaches exist for the vertical fragmentation of global relations:

1. Grouping: starts by assigning each attribute to one fragment, and at each step, joins some of the fragments until some criteria is satisfied. Grouping was first suggested in [Hammer and Niamey, 1979] for centralized databases, and was used later in [Sacra and Wielder-hold, 1985] for distributed databases.

2. Splitting: starts with a relation and decides on beneficial partitioning based on the access behavior of applications to the attributes. The technique was first discussed for centralized database design in [Hoofer and Severance, 1975]. It was then extended to the distributed environment in [Nava the et al., 1984].

In what follows we discuss only the splitting technique, since it fits more naturally within the top-down design methodology, and as stated in [Nava the et al., 1984], since the "optimal" solution is probably closer to the full relation than to a set of fragments each of which consists of a single attribute. Furthermore, splitting generates non overlapping fragments whereas grouping typically results in overlapping fragments. Within the context of distributed database systems, we are concerned with non

overlapping fragments, for obvious reasons. Of course, none overlapping refers only to non primary key attributes.

There is a strong advantage to replicating the key attributes despite the obvious problems it causes. If we now design the database so that the key attributes are part of one fragment that is allocated to one site, and the implied attributes are part of another fragment that is allocated to a second site, every update request that causes an integrity check will necessitate communication among sites. Replication of the key attributes at each fragment reduces the chances of this occurring but does not eliminate it completely, since such communication may be necessary due to integrity constraints that do not involve the primary key, as well as due to concurrency control.

One alternative to the replication of the key attributes is the use of topple identifiers (TIDs), which are system-assigned unique values to the topples of a relation. Since TIDs are maintained by the system, the fragments are disjoint as far as the user is concerned.

Information requirements of vertical fragmentation: The major information required for vertical fragmentation is related to applications. The following discussion, therefore, is exclusively on what needs to be determined about applications that will run against the distributed database. Since vertical partitioning places in one fragment those attributes usually accessed together, there is a need for some measure that would define more precisely the notion of "togetherness." This measure is the affinity of attributes, which indicates how closely, related the attributes are. Unfortunately, it is not realistic to expect the designer or the users to be able to easily specify these values. We now present one way by which they can be obtained from more primitive data.

4.3.3 Hybrid Fragmentation

In most cases a simple horizontal or vertical fragmentation of a database schema will not be sufficient to satisfy the requirements of user applications. In this case a vertical fragmentation may be followed by a horizontal one, or vice versa, producing a tree structured partitioning. Since the two types of partitioning strategies are applied one after the other, this alternative is called hybrid fragmentation. It has also been named mixed fragmentation or nested fragmentation.

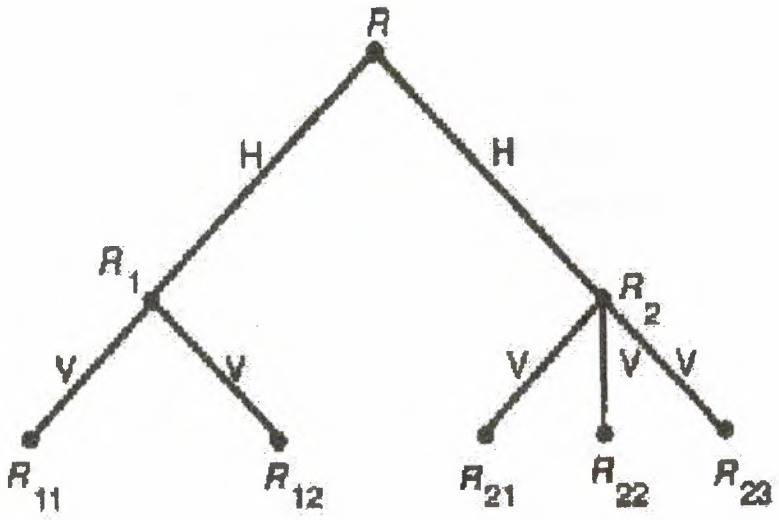

Figure 4.10: Hybrid Fragmentation

A good example for the necessity of hybrid fragmentation is relation J, which we have been working with. What we have, therefore, is a set of horizontal fragments, each of which is further partitioned into two vertical fragments.

The number of levels of nesting can be large, but it is certainly finite. In the case of horizontal fragmentation, one has to stop when each fragment consists of only one tuple, whereas the termination point for vertical fragmentation is one attribute per fragment. These limits are quite academic, however, since the levels of nesting in most practical applications do not exceed 2. This is due to the fact that normalized global relations already have small degrees and one cannot perform too many vertical fragmentations before the cost of joins becomes very high.

We will not discuss in detail the correctness rules and conditions for hybrid fragmentation, since they follow naturally from those for vertical and horizontal fragmentations. For example, to reconstruct the original global relation in case of hybrid fragmentation, one starts at the leaves of the partitioning tree and moves upward by performing joins and unions. The fragmentation is complete if the intermediate and leaf fragments are complete. Similarly, disjoint ness is guaranteed if intermediate and leaf fragments are disjoint.

4.4 ALLOCATION

The allocation of resources across the nodes of a computer network is a problem that has been studied extensively. Most of this work, however, does not address the problem of distributed database design, but rather that of placing individual files on a computer network. We will examine the differences between the two shortly. We first need to define the allocation problem more precisely.

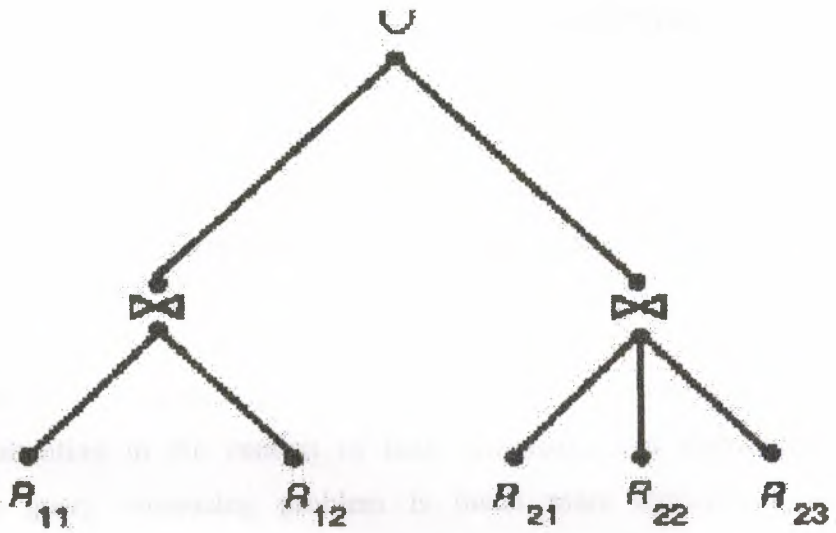

Figure 4.11: Reconstruction of Hybrid Fragmentation

CHAPTER FIVE

QUERY PROCESSING

The increasing success of relational database technology in data processing is due, in part, to the availability of nonprocedural languages, which can significantly improve application development and end-user productivity. By hiding the low-level details about the physical organization of the data, relational database languages allow the expression of complex queries in a concise and simple fashion. In particular, to construct the answer to the query, the user does not precisely specify the procedure to follow. This procedure is actually devised by a DBMS module, usually called a query processor. This also relieves the user from query optimization, a time consuming task that is best handled by the query processor, since it can exploit a large amount of useful information about the data.

Because it is a critical performance issue, query processing has received considerable attention in the context of both centralized and distributed DBMSs. However, the query processing problem is much more difficult in distributed environments than in centralized ones, because 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 context chosen is that of relational calculus and relational algebra, because of their generality and wide use in distributed DBMSs, distributed relations are implemented by fragments. Distributed database design is of major importance for query processing since the definition of fragments is based on the objective of increasing reference locality, and sometimes parallel execution for the most important queries. The role of a distributed query processor is to map a high-level query (assumed to be expressed in relational calculus) on a distributed database (i.e., a set of global relations) into a sequence of database operations (of relational algebra) on relation fragments. Several important functions characterize this mapping. First, the calculus query must be decomposed into a sequence of relational operations called an algebraic query. Second, the data accessed by the query must be localized so that the operations on relations are translated to bear on local data (fragments). Finally, the algebraic query

on fragments must be extended with communication operations and optimized with respect to a cost function to be minimized. This cost function typically refers to computing resources such as disk I/Os, CPUs, and communication networks.

5.1 QUERY PROCESSING PROBLEM

The main function of a relational query processor is to transform a high-level query (typically, in relational calculus) into an equivalent lower-level query (typically, in some variation of relational algebra). The low-level query actually implements the execution strategy for the query. The transformation must achieve both correctness and efficiency. It is correct if the low-level query has the same semantics as the original query, that is, if both queries produce the same result. The well-defined mapping from relational calculus to relational algebra makes the correctness issue easy. But producing an efficient execution strategy is more involved. A relational calculus query may have many equivalent and correct transformations into relational algebra. Since each equivalent execution strategy can lead to very different consumptions of computer resources, the main difficulty is to select the execution strategy that minimizes resource consumption.

Example 7.1

E(ENO, ENAME, TITLE)

G (ENO, JNO, RESP, DUR)

And the following simple user query:

"Find the names of employees who are managing a project"

The expression of the query in relational calculus using the SQL syntax is

SELECT ENAME

FROM E, G

WHERE E.ENO = G.ENO

AND RESP = “Manager”

Two equivalent relational algebra queries that are correct transformations of the query above are

$\Pi_{ENAME}(\sigma_{RESP=\text{"Manager"} \wedge E.ENO=G.ENO}(E \times G))$

and

$\Pi_{ENAME}(E \bowtie_{ENO} (\sigma_{RESP=\text{“Manager”}}(G)))$

It is intuitively obvious that the second query, which avoids the Cartesian product of E and G, consumes much less computing resource than the first and thus should be retained.

In a centralized context, query execution strategies can be well expressed in an extension of relational algebra. The main role of a centralized query processor is to choose, for a given query, the best relational algebra query among all equivalent ones. Since the problem is computationally intractable with a large number of relations [Ibaraki and Kameda, 1984], it is generally reduced to choosing a solution close to the optimum.

In a distributed system, relational algebra is not enough to express execution strategies. It must be supplemented with operations for exchanging data between sites. Besides the choice of ordering relational algebra operations, the distributed query processor must also 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, making distributed query processing significantly more difficult.

5.2 OBJECTIVES OF QUERY PROCESSING

As stated before, the objective of query processing in a distributed context is to trans form a high-level query on a distributed database, which is seen as a single database by the users, into an efficient execution strategy expressed in a low-level language on local databases. We assume that the high-level language is relational calculus, while the low-level language is an extension of relational algebra with communication operations. The different layers involved in the query transformation are detailed in Section 7.5. An important aspect of query processing is query optimization. Because many execution strategies are correct transformations of the same high-level query, the one that optimizes (minimizes) resource consumption should be retained.

A good measure of resource consumption is the total cost that will be incurred in processing the query. Total cost is the sum of all times incurred in processing the operations of the query at various sites and in inter site communication. Another good measure is the response time of the query, which is the time elapsed for executing the

query. Since operations can be executed in parallel at different sites, the response time of a query may be significantly less than its total cost.

In a distributed database system, the total cost to be minimized includes CPU, I/O, and communication costs. The CPU cost is incurred when performing operations on data in main memory. The I/O cost is the time necessary for disk input/output operations. This cost can be minimized by reducing the number of I/O operations through fast access methods to the data and efficient use of main memory (buffer management). The communication cost is the time needed for exchanging data between sites participating in the execution of the query. This cost is incurred in processing the messages (formatting/ de formatting), and in transmitting the data on the communication network.

The first two cost components (I/O and CPU cost) are the only factors considered by centralized DBMSs. The communication cost component is probably the most important factor considered in distributed databases. Most of the early proposals for distributed query optimization assume that the communication cost largely dominates local processing cost (I/O and CPU cost), and thus ignore the latter. This assumption is based on very slow communication networks (e.g., wide area networks with a bandwidth of a few kilobytes per second) rather than on networks with disk bandwidths. Therefore, the aim of distributed query optimization is simplified to the problem of minimizing communication costs generally at the expense of local processing. The advantage is that local optimization can be done independently using the known methods for centralized systems. However, distributed processing environments now exist where the communication network is much faster (e.g., local area networks) and that can have a bandwidth comparable to that of disks. Therefore, more recent research efforts consider a weighted combination of these three cost components since they all contribute significantly to the total cost of evaluating a query. Nevertheless, in distributed environments with high bandwidths, the overhead cost incurred for communication between sites (e.g., software protocols) makes communication cost still an important factor as important as I/O cost. For completeness, let us consider the methods that minimize all cost components.

5.3 CHARACTERIZATION OF QUERY PROCESSORS

It is quite difficult to evaluate and compare query processors in the context of both centralized systems and distributed systems because they may differ in many aspects. In what follows, we list important characteristics of query processors that can be used as a basis for comparison. The first four characteristics hold for both centralized and distributed query processors, while the next four characteristics are particular to distributed query processors.

5.3.1 Languages

Initially, most work on query processing was done in the context of relational databases because their high-level languages give the system many opportunities for optimization. The input language to the query processor can be based on relational calculus or relational algebra. The former requires an additional phase to decompose a query expressed in relational calculus into relational algebra. In a distributed context, the output language is generally some internal form of relational algebra augmented with communication primitives.

The operations of the output language are implemented directly in the system. Query processing must perform efficient mapping from the input language to the output language.

5.3.2 Types of Optimization

Conceptually, query optimization aims at choosing the best point in the solution space of all possible execution strategies. An immediate method for query optimization is to search the solution space, exhaustively predict the cost of each strategy, and select the strategy with minimum cost. Although this method is effective in selecting the best strategy, it may incur a significant processing cost for the optimization itself. The problem is that the solution space

Can be large; that is, there may be many equivalent strategies, even with a small number of relations. The problem becomes worse as the number of relations increases. Having high optimization cost is not necessarily bad, particularly if query optimization is done once for many subsequent executions of the query. Therefore, the exhaustive search approach is often used [Salinger et al., 1979].

One popular way of reducing the cost of exhaustive search is the use of heuristics, whose effect is to restrict the solution space so that only a few strategies are considered. In both centralized and distributed systems, a common heuristic is to minimize the size of intermediate relations. This can be done by performing unary operations first, and ordering the binary operations by the increasing sizes of their intermediate relations. An important heuristic in distributed systems is to replace join operations by combinations of semi joins to minimize data communication.

5.3.3 Optimization Timing

A query may be optimized at different times relative to the actual time of query execution. Optimization can be done statically before executing the query or dynamically as the query is executed. Static query optimization is done at query compilation time. Thus the cost of optimization may be amortized over multiple query executions.

Therefore, this timing is appropriate for use with the exhaustive search method. Since the sizes of the intermediate relations of a strategy are not known until run time, they must be estimated using database statistics. Errors in these estimates can lead to the choice of sub optimal strategies.

Dynamic query optimization proceeds at query execution time. At any point of execution, the choice of the best next operation can be based on accurate knowledge of the results of the operations executed previously. Therefore, database statistics are not needed to estimate the size of intermediate results. However, they may still be useful in choosing the first operations. The main advantage over static query optimization is that the actual sizes of intermediate relations are available to the query processor, thereby minimizing the probability of a bad choice. The main shortcoming is that query optimization, an expensive task, must be repeated for each execution of the query.

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

5.3.4 Statistics

The effectiveness of query optimization relies on statistics on the database. Dynamic query optimization requires statistics in order to choose which operations should be done first. Static query optimization is even more demanding since the size of intermediate relations must also be estimated based on statistical information. In a distributed database, statistics for query optimization typically bear on fragments, and include fragment cardinality and size as well as the size and number of distinct values of each attribute. To minimize the probability of error, more detailed statistics such as histograms of attribute values are sometimes used at the expense of higher management cost. The accuracy of statistics is achieved by periodic updating. With static optimization, significant changes in statistics used to optimize a query might result in query re optimization.

5.3.5 Decision Sites

When static optimization is used, either a single site or several sites may participate in the selection of the strategy to be applied for answering the query. Most systems use the centralized decision approach, in which a single site generates the strategy. However, the decision process could be distributed among various sites participating in the elaboration of the best strategy. The centralized approach is simpler but requires knowledge of the entire distributed database, while the distributed approach requires only local information. Hybrid approaches where one site makes the major decisions and other sites can make local decisions are also frequent. For example, R* [Williams et al., 1982] uses a hybrid approach.

5.3.6 Exploitation of the Network Topology

The network topology is generally exploited by the distributed query processor. With wide area networks, the cost function to be minimized can be restricted to the data communication cost, which is considered to be the dominant factor. This assumption greatly simplifies distributed query optimization, which can be divided into two separate problems: selection of the global execution strategy, based on inter site communication, and selection of each local execution strategy, based on a centralized query processing algorithm.

With local area networks, communication costs are comparable to I/O costs. Therefore, it is reasonable for the distributed query processor to increase parallel execution at the expense of communication cost. The broadcasting capability of some local area networks can be exploited successfully to optimize the processing of join operations ([Ozsoyoglu and Zhou, 1987] and [Wahl and Lien, 1985]). Other algorithms specialized to take advantage of the network topology are presented in [Kerschberg et al., 1982] for star networks and in [LaChimia, 1984] for satellite networks.

5.3.7 Exploitation of Replicated Fragments

Distributed queries expressed on global relations are mapped into queries on physical fragments of relations by translating relations into fragments. We call this process localization because its main function is to localize the data involved in the query. For reliability purposes it is useful to have fragments replicated at different sites. Most optimization algorithms consider the localization process independently of optimization. However, some algorithms exploit the existence of replicated fragments at run time in order to minimize communication times. The optimization algorithm is then more complex because there are a larger number of possible strategies.

5.3.8 Use of Semi joins

The semi join operation has the important property of reducing the size of the operand relation. When the main cost component considered by the query processor is communication, a semi join is particularly useful for improving the processing of distributed join operations as it reduces the size of data exchanged between sites. However, using semi joins may result in an increase in the number of messages and in the local processing time. The early distributed DBMSs, such as SDD-1 [Bernstein et al., 1981], which were designed for slow wide area networks, make extensive use of semi joins. Some recent systems, such as R* [Williams et al., 1982], assume faster networks and do not employ semi joins. Rather, they perform joins directly since using joins leads to lower local processing costs. Nevertheless, semi joins are still beneficial in the context of fast networks when they induce a strong reduction of the join operand, therefore, some recent query processing algorithms aim at selecting an optimal combination of joins and semi joins.

CONCLUSION

Distributed database system (DDBS) technology is the union of what appear to be two diametrically opposed approaches to data processing: database system and computer network technologies. Database systems have taken us from a paradigm of data processing, in which each application defined and maintained its own data to one in which the data is and administered centrally. This new orientation results in data independence, whereby the application programs are immune to changes in the logical or physical organization of the, and vice versa.

I presented the techniques that can be used for Distributed Database Design with special emphasis on the fragmentation and allocation issues. There are a number of lines of research that have been followed in distributed daatabase design.For example, Chang has independently developed a theory of fragmentation [Chang and Cheng, 1980], and allocation [Chang and Liu, 1982]. However, for its maturity of development, we have chosen to develop this chapter along the track developed by Ceri, Pelagatti Navathe, and Wiederhold. Our references to the literature by these authors reflect this quite clearly.

I hope that this project will be usefully for both future life and other people who are interested in DISTRIBUTED DATABASE SYSTEM.

REFERENCE

1. A. E. Abbadi, D. Skeen, and F. Cristian. An Efficent, Faculty-Tolerant Protocol for Replicated Data Management.In Proc.4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, Portland, Oreg.,March 1985, pp. 215-229.

2. American National Standart for Information System. Database Language SQL. ANSI X3.135-1986, October 1986.

3. K. Barker and M.T.Özsu. Survey of Issues in Distributed Heterogeneous Database Systems. Technical Report TR88-9, Edmonton, Alberta, Canada: Department of Computing Science. University of Alberta 1988.

4. G. von Bochmann. Concepts for Distributed Systems Design. Berlin: Springer-Verlag, 1983.

5. P. Valduriez. Join Indices. ACM Trans. Database System (June 1987).

6. B. W. Wah and Y. N. Lien. Design of Distributed Databases on Local Computer Systems. IEEE Trans. Software Eng. (July 1985),

7. B. Yormark. The ANSI/SPARC/DBMS Architecture. In ANSI/SPARC/DBMS Model, D. A. Jardine (ed.), Amsterdam: North-Holland, 1977.

8. http://www.fbe.itu.edu.tr/docs/info/cources/kontbil.html