

**ON TRANSPARENCY IN THE DISTRIBUTED
DATABASE SYSTEMS**

Dissertation Submitted to
JAWAHARLAL NEHRU UNIVERSITY
in partial fulfilment of requirements
for the award of the degree of

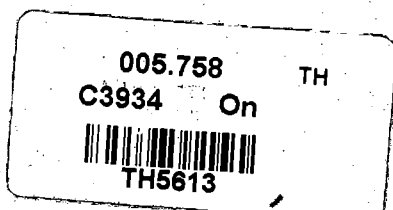
Master of Technology
in
Computer Science

by
K.S. Chaudhary



SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067


January 1996

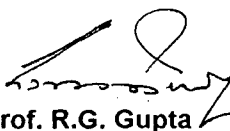


CERTIFICATE

This is to certify that the dissertation entitled
ON TRANSPARENCY IN THE DISTRIBUTED DATABASE SYSTEMS
which is being submitted by Kirti Singh Chaudhary to the School of
Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi
for the award of Master of Technology in Computer Science, is a record of
bonafide project work carried out by him under the supervision and guidance
of
Prof. R.G. Gupta.

This work is original and has not been submitted in part or full to any
University or Institution for the award of any degree.


Prof. G.V. Singh
(Dean, SC & SS)


Prof. R.G. Gupta
(Supervisor)

ACKNOWLEDGEMENT

I wish to convey my heartfelt gratitude and sincere acknowledgements to my guide *Prof R.G. Gupta*, School of Computer & Systems Sciences for his whole hearted, tireless and relentless effort in helping me for the successful completion of the project.

I would like to record my sincere thanks to my Dean, *Prof. G. V. Singh*, School of Computer & Systems Sciences for providing the necessary facilities in the centre for the successful completion of the project.

I take this opportunity to thank all my faculty members and friends for their critical comments during the course of the project.

Kirti Singh Chaudhary

CONTENTS

1. Introduction	1
1.1. Why Go For Distributed Database	1
1.2. Feature of Distributed Verses Centralized Databases.....	2
2. Design Methodology in the Distributing Data in Distributed Databases	6
2.1. Introduction.....	6
2.2. Component of a Distributed Database Design.....	7
2.3. Objectives of the Design of Data Distribution.....	7
2.4. Distributed Data Design Methodology.....	9
2.5. Data Fragmentation.....	10
3. Functions of Distributed DBMS and Transparencies.....	14
3.1. Reference Architecture for Distributed Databases.....	15
3.2. Functions of DDBMS.....	18
3.3. Distribution Transparency for Read-Only applications: An Example.....	20
4. Use of Transparency in View of Distributed Databases.....	24
4.1. Introduction.....	24
4.2. Views in Distributed Database System.....	25
4.3. Distributed View Management.....	28
5. Pros and Cons of Transparency.....	31
5.1. A Case Against Transparent Access to Distributed Data.....	32
5.2. Manageability of a Distributed Database.....	34
5.3. The Case for Transparent Access to Cluster Data.....	34
6. R*: An Example of Location Transparency.....	36
6.1. Problems Solved By R* System.....	38
6.2. View Management in R*.....	39
6.3. R* Objectives.....	40
7. Conclusion.....	43
Bibliography.....	44

1 INTRODUCTION

From a technical viewpoint, distributed database technology is the marriage of computer communication networking with database technology. From a user viewpoint, it is an application of computer communication networking and provides an efficient means of showing databases. Today there is a need for the construction of distributed databases for the geographically separated user community. Before going in depth of distributed databases, we will discuss the centralized databases and the distributed databases in comparison of each other.

1.1. Why Go For Distributed Database

The basic premise behind a distributed database system is to make a data store spread over a collection of machines in a wide or local area network appear to the user as if it existed on a single machine.

As companies are fundamentally distributed, these have many offices in many places, there is every incentive for each location to have its own computer. If for no other reason, the cost of communication is prohibitive. One simply does not want to have a terminal in Hong Kong that accesses a computer in New York; one will quickly have a gigantic communications bill. All companies having computer systems, typically spread around the world, and there are databases on each of these machines, and most organizations must run applications that access data in multiple databases on multiple computer systems. A simple example of such geographically distributed data is the INGRES division of ASK. They maintain a customer database for the USA customers at their headquarter in California. In addition the British customers are on the machine in Frankfurt, the French customers are on the machine in London, the German customers are on the machine in Frankfurt, the French customers are on the machine in Paris, and so on. In general, there is high locality of reference to the customer database; hence, the French

customers are usually accessed by the French INGRES personnel. However, there are also across database accesses. For example, sometimes a salesman wants to know all the INGRES customer in the world that are part of some multinational corporation. This requires access to individual customer databases at all INGRES locations. Obviously, a distributed databases that provides so called **location transparency** will expedite such multidatabase queries.

However, geographical distribution is not the only reason to have distributed database system. It is obvious that dumb terminals will be replaced on most people desks by workstations. Moreover, most workstations have local disks to achieve the factor of these in performance that can be obtained. Moreover, a DBMS will run on each workstation and manage local databases. Such databases will at least contain personal data such as telephone directories, appointment calendars etc. In such an environment one would clearly like to run a program that would schedule a meeting by intersecting the available times for all the participants. Obviously, this is a distributed database application. Hence the imminent presence of large no. of workstations with disks will create the need for a distributed database system.

1.2. Feature of Distributed Verses Centralized Databases

Here it is required to look at the typical features of centralized databases and to compare them with the corresponding features of distributed databases. The features which characterize the centralized database approach are centralized control, data independence, reduction of redundancy, complex physical structure for efficient access, integrity, recovery, concurrent control, privacy and security.

Centralized Control In centralized databases, the centralized control over information resources is considered as one of the strongest feature. The fundamental function of a DBA is to guarantee the safety of data which requires a centralized responsibility.

In distributed databases the idea of centralized control is much less emphasized and also depends upon the various architecture and transparency. In general, in distributed databases it is possible to identify a hierarchical control structure based on a global database administrator, who has the control responsibility of the whole database and on local database administrators who have the responsibility of their respective local databases. However, it must be emphasized that local database administrator may have a high degree of autonomy upto the point that a global database administrator is completely missing and the intersite coordination is performed by the local administrative themselves. This characteristic is usually called Site Autonomy.

Data Independence In Distributed databases, data independence has the same importance as in centralized databases, however, a new aspect is added to the usual notion of data independence, namely **distribution transparency**. By distribution transparency we mean that program can be unaffected by the movement of data from one site to another, however, their speed of execution is affected.

Data independence is provided in centralized databases through a multilevel architecture having different descriptions of data and mappings between them. The notion of conceptual schema, storage schema and external schemas were developed for this purpose. In the same way distribution transparency is obtained in distributed databases by introducing new levels and schemata.

Reduction of Redundancy In centralized databases, redundancy was reduced as far as possible so that inconsistencies among several copies of the same logical data are automatically avoided by having only one copy and also storage space is saved.

In Distributed databases, however there are several reasons for considering data redundancy:

i) locality of applications can be increased if the data is replicated at all sites where applications need it.

ii) availability of the system can be increased because a site failure does not stop the execution of applications at other sites if the data is replicated.

The convenience of data replication increases because if we have several copies of an item, retrieval can be performed on any copy, while updates must be performed consistently on all copies.

Complex Physical Structures and Efficient Access Complex accessing
structures like secondary indexes interfile chains are a major aspect of centralized databases. The reason is to obtain efficient access to the data.

In distributed databases, complex accessing structures are not the right tool for efficient access. Therefore, while efficient access is a main problem in distributed databases, physical structures are not a relevant technological issue. Efficient access to a distributed databases cannot be provided by using intersite physical structures, because it is very difficult to build and maintain such structures.

Integrity, Recovery and Concurrency Control Integrity and recovery are major aspects which are particularly important in distributed databases because some of the sites involved in transaction execution might fail. Concurrency control deals with ensuring transaction atomicity in the presence of concurrent execution of transactions. This problem can be seen as a typical synchronisation problem. In distributed databases, the synchronisation problem is harder than in centralized system.

Privacy and Security In centralized databases, the DBA, having centralized control can ensure that only authorized access to the data is performed. In distributed databases, local administrators are faced essentially with the same problem as DBA in centralized. However two peculiar aspects of distributed databases are:-

(i) in a distributed database with a very high degree of autonomy the owners of local data feel more protected because they can enforce their own protections instead of depending on a central DBA.

(ii) security problems are intrinsic to distributed systems in general because communication networks can represent a weak point w.r.t. protection.

Thus we have seen that distributed database technology extends that centralized database technology in a nontrivial way, and in brief advantages of a distributed database over a centralized one are increased availability decreased access time, easy expansion and possible integration of existing databases. The acceptance and widespread usage of distributed databases will highly depend on their efficiency. Therefore it is important to supply a database management system with tools to efficiently process queries and to determine allocation of the data such that the availability is increased, the access time is decreased and/or the overall usage of resources is minimized.

2 DESIGN METHODOLOGY IN DISTRIBUTING THE DATA IN DISTRIBUTED DATABASES

2.1. Introduction

Design methodology in distributed data includes the allocation of data and operations to nodes in a computer communications network together with the transparency of the database is a critical issue in distributed database design. An efficient design must trade off performance and cost among retrieval and update activities at the various nodes. It must consider the concurrency control mechanism used as well as capacity constraints at nodes and on links in the network. It must determine where data will be allocated, the degree of data replication, which copy of the data will be used for each retrieval activity and where operations such as SELECT, PROJECT, JOIN and UNION will be performed.

In general, satisfying a user request in a distributed database involves five major steps:

- i) Accessing the network directory to determine where the needed data is located.
- ii) Determining an access strategy that specifies which copy of the data to access (and when), where the data will be processed, and how it will be routed.
- iii) Sending request messages to the appropriate nodes(for update requests this may involves waiting for locks to be established and response messages to be sent back).
- iv) Accessing and processing data at each of these nodes, and
- v) Routing the response to the requesting node for final processing.

If (a copy of) all the data required by a retrieval request is located at the requesting node, then only local accessing and processing are needed. However, if

some needed data is not located at the requesting node, then data must be accessed from, and possibly processed at other nodes.

Key distributed design issues include data and operation allocation. Data allocation defines what data is stored at what nodes (including possible redundancy). Operation allocation defines where accessing and processing operations will be performed.

2.2. Component of a Distributed Database Design

Design of distributed database systems is extremely complex. There are four major aspects which must be considered

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

2.3. Objectives of the Design of Data Distribution

In the design of data distribution the following objectives should be taken into consideration:

Processing locality : Distributing data to maximize processing locality corresponds to the simple principle of placing data as close as possible to the applications which use them. The simplest way of characterizing processing locality is to consider two types of references to data: "local" references and "remote" references. Clearly, once the sites of origin of applications are known, locality and remoteness of references depend only on the data distribution.

Designing data distribution for maximizing processing locality can be done by adding the no. of local and remote references corresponding to each candidate fragmentation and fragment allocation and selecting the best solution among them.

An extension to this simple optimization criterion is to take into account when an application has complete locality, which can be completely executed at their sites of origin. The advantage of complete locality is not only the reduction of remote accesses, but also the increased simplicity in controlling the execution of the application.

Availability and Reliability of Distributed Data : A high degree of availability for read-only application is achieved by storing multiple copies of the same information; the system must be able to switch to an alternative copy when the one that should be accessed under normal conditions is not available.

Reliability is also achieved by storing multiple copies of the same information, since it is possible to recover from crashes or from the physical destruction of one of the copies by using the other, still available copies. Since physical destruction can be caused by events which have nothing to do with computer crashes, it is relevant to store replicated copies in geographically dispersed locations.

Workload distribution : Distributing the workload over the sites is an important feature of distributed computer systems. Workload distribution is done in order to take advantage of the different powers or utilization of computers at each site and to maximize the degree of parallelism of execution of applications. Since workload locality, it is necessary to consider the trade-off between them in the design of data distribution.

Storage costs and availability : Database distribution should reflect the cost and availability of storage at different sites. It is possible to have specialized sites in the network for data storage, or conversely to have sites which do not support mass storage at all. Typically the cost of data of data storage is not relevant if compared

with CPU, I/O and transmission costs of applications, but the limitation of available storage at each site must be considered.

2.4. Distributed Data Design Methodology

The distributed data design methodology given in Fig. 1, takes into consideration data distribution requirements. The methodology produces a global conceptual schema and local logical and physical schemata for each network site. A new *data distribution design* step in the methodology takes in the global conceptual schema and the distribution requirements of the system. This step performs two principal activities. First, a data allocation design for the distributed system is developed. Then, based on the data allocation, the global conceptual schema is divided into local conceptual schemata for each of the processing sites in the system. This step in the methodology passes the local conceptual schemata to each site where the logical and physical data design steps are performed.

Four significant design decisions compose the data allocation design for distributed systems:

1. *Data Partitioning*. Partitioning (or, fragmentation) is the process of determining divisions of data to be used as allocation units on the network. A given data entity can be divided either vertically or horizontally. *Vertical partitions* divide an entity by grouping attributes into defined subsets. Each partition must contain a key attribute that identifies the unique entity occurrence associated with each record. This is needed in order to rebuild a complete entity across all vertical partitions. *Horizontal partitions* divide an entity by grouping records based on some qualification on attribute values. The original entity can be regained by unioning the records from all horizontal partitions.

2. *Data Placement*. Placement determines at which site in the distributed system each data partition is placed for best benefit.

3. *Data Replication*. Replication determines how many copies of each data partition are placed on the system at different sites. The resulting data redundancy

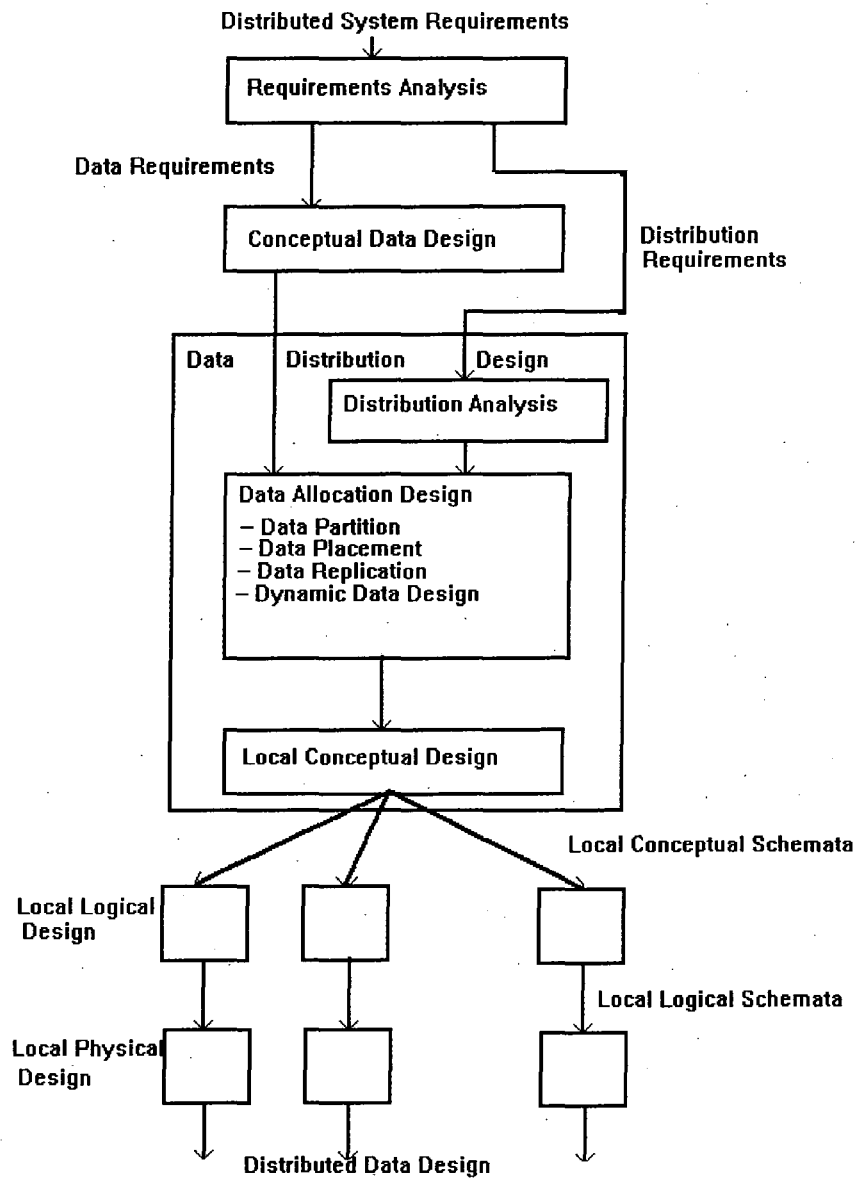


Fig. 1. Distributed Data Design Methodology

provides some of the major benefits of distributed systems in data availability and reliability. However, the costs of data replication are significant. Decisions on data placement and replication are closely related and the decision processes for placement and replication are highly interactive.

4. *Dynamic Data Allocation.* Initial distributed data designs may turn out to be unsatisfactory when the system is in operation, either because of poor design decisions or changes in system requirements. A strategy of monitoring system performance and altering the distributed data allocation is an important component of a total system data design. While for some systems the answer may be to stop the system and perform a complete reallocation of the distributed data, many systems will require some ability to dynamically perform data partitioning, data placement and data replication. A decision process for invoking these dynamic operations is needed.

2.5. Data Fragmentation

The first step of a distributed data allocation is determining the most effective partitions of data to be units of distribution. We term this the *data fragmentation* problem. Previously, designers have simply taken the data file to be the unit of allocation. However, several performance advantages can be gained by partitioning data files into smaller units for allocation. Vertical fragmentation or partitioning group attributes that have a high probability of being accessed together, but at the same time keeping the partitions group records of an entity that satisfy a given condition on attribute values. For example, a large personnel file can be partitioned based upon the department in which an employee works. Each horizontal fragmentation or partition can then be distributed to the site of that department.

The shared database should be fragmented so that the system throughput is maximal. The fragmenting of the database are executed efficiently, while the capacity constraints of all the processors and limits of the communication network is observed. A poor partitioning of the database can lead to higher processing costs

in the nodes or to greater demands on the communication network, so that the system may not be able to handle the required set of transactions.

The database resides on multiple processors which are connected using a dense and reliable communication network. The contents of the database are assigned to the processor nodes. Associated with every transaction is an initial network node. Since the user is not connected to any prespecified node, the initial network node is to be assigned as part of the design.

The capacity limit of each of the processors may be given in terms of processor cycle capacity and IO-block move capacity per unit time. The capacity limit of the network is given in terms of aggregate block transfer capacity per unit time. The demands of the transactions are then given using the same measures.

As we know the database is a collection of relations. The given conceptual relation may be too large to be effectively assigned to single processors. We initially consider how the relation can be fragmented, and then allocate those fragments to the processor nodes. In the designed database, each transaction accesses some subset of the tuples and the attributes of each original relation. In order to complete transactions that do not find all their data on the same processor, a scheduling algorithm can be invoked which can optimize the processing over the network.

There are, however, some rules which must be followed when defining fragments:

Completeness condition : All the data of the global relation must be mapped into the fragments; i.e., it must not happen that a data item which belongs to a global relation does not belong to any fragment.

Reconstruction condition : It must always be possible to reconstruct each global relation from its fragments. The necessity of this condition is obvious: in fact, only fragments are stored in the distributed database, global relations have to be built through this reconstruction operation, if necessary.

Disjointness condition : This condition is useful mainly with horizontal fragmentation, while for vertical fragmentation we should sometimes allow this condition to be violated.

We can now discuss the fragmentation rules.

Horizontal Fragmentation Horizontal fragmentation consists of partitioning the tuples of global relation into subsets; this is clearly useful in distributed databases, where each subset can contain data which have common geographical properties. It can be defined by expressing each fragment as a selection operation on the global relation. For example, let a global relation be

SUPPLIER(SNUM, NAME, CITY)

Then the horizontal fragmentations can be defined in the following way:

SUPPLIER₁ = $\sigma_{CITY = "SF"}$ SUPPLIER

SUPPLIER₂ = $\sigma_{CITY = "LA"}$ SUPPLIER

The above fragmentation satisfies the completeness condition if "SF" and "LA" are the only possible values of the CITY attribute; otherwise we would not know to which fragment the tuples with other CITY values belong.

Vertical Fragmentation The vertical fragmentation of a global relation is the subdivision of its attribute into groups; fragments are obtained by projecting the global relation over each group. This can be useful in distributed databases where each group of attributes can contain data which have common geographical properties. The fragmentation is correct if each attribute is mapped into at least one attribute of the fragments; moreover, it must be possible to reconstruct the original relation by joining the fragments together. Consider, for example, a global relation

EMP(EMPNUM, NAME, SAL, TAX, MGRNUM, DEPTNUM)

A vertical fragmentation of this relation can be defined as

EMP₁ = $\rho_{EMPNUM, NAME, MGRNUM, DEPTNUM}$ EMP

$$EMP_2 = PJ_{EMPNUM, SAL, TAX} EMP$$

This fragmentation could reflect an organization in which salaries and taxes are managed separately. The reconstruction of relation *EMP* can be obtained as

$$EMP = EMP_1 JN_{EMPNUM} = EMPNUM EMP_2$$

because *EMPNUM* is a key of *EMP*.

3

FUNCTIONS OF DISTRIBUTED DBMS WITH RESPECT TO TRANSPARENCIES

The basic premise behind a distributed database system is to make a data store spread over a collection of machines in a wide or local area network appear to the user as if it existed on a single machine.

As companies are fundamentally distributed and these have many offices in many places, and there is every incentive for each location to have its own computer. If for no other reason, the cost of communication is prohibitive. One simply does not want to have a terminal in Hong Kong that accesses a computer in New York; one will quickly have a gigantic communications bill. All companies having computer systems, typically spread around the world, and there are databases on each of these machines, and most organizations must run applications that access data in multiple databases on multiple computer systems. A simple example of such geographically distributed data is the INGRES division of ASK. They maintain a customer database for the USA customers at their headquarter in California. In addition the British customers are on the machine in Frankfurt, the French customers are on the machine in London, the German customers are on the machine in Frankfurt, the French customers are on the machine in Paris, and so on. In general, there is high locality of reference to the customer database; hence, the French customers are usually accessed by the French INGRES personnel. However, there are also across database accesses. For example, sometimes a salesman wants to know all the INGRES customer in the world that are part of some multinational corporation. This requires access to individual customer databases at all INGRES locations. Obviously, a distributed databases that provides so called **location transparency** will expedite such multidatabase queries.

However, geographical distribution is not the only reason to have distributed database system. It is obvious that dumb terminals will be replaced on most people desks by workstations. Moreover, most workstations have local disks to achieve the factor of these in performance that can be obtained. Moreover, a DBMS will run on each workstation and manage local databases. Such databases will at least contain personal data such as telephone directories, appointment calendars etc. In such an environment one would clearly like to run a program that would schedule a meeting by intersecting the available times for all the participants. Obviously, this is a distributed database application. Hence the imminent presence of large no. of workstations with disks will create the need for a distributed database system.

Now we will discuss the different levels at which an application programmer views the distributed database, depending on how much distribution transparency is provided by the distributed database management system. Distribution transparency is nothing but the independence of the application program from the distribution of data and has been considered to be conceptually equivalent to data independence in centralized databases.

There are several levels of distribution transparency. For studying these levels, we considered a layered reference architecture for a distributed database. This architecture will allow us to determine easily different levels of distribution transparencies. The levels are conceptually relevant in order to understand distributed databases.

3.1. Reference Architecture for Distributed Databases

Fig. 2 shows a reference architecture for a distributed database. This reference architecture is not explicitly implemented in all distributed databases; however, its levels are conceptually relevant in order to understand the organization of any distributed databases.

At the top of the level is the global schema. The **global schema** defines all the data which are contained in the distributed database as if the database were not

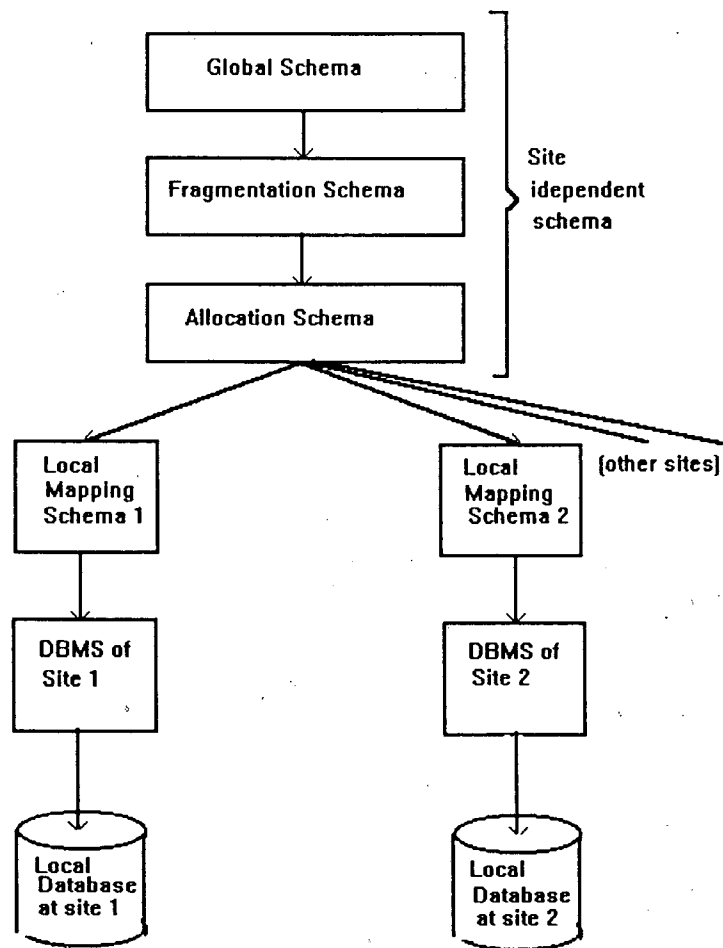


Fig. 2: A Reference Architecture for Distributed Databases

distributed at all. For this reason, the global schema can be defined exactly in the same way as in a non distributed database. However, the data model which is used for the definition of a global schema should be convenient for the definition of the mapping to the other levels of the distributed database. Using this model, the global schema consists of the definition of a set of **global relations**.

Each global relation can be split into several nonoverlapping portions which are called **fragments**. There are several different ways in which to perform the splitting operations. The mapping between global relations and fragments is defined in the **fragmentation schema**. This mapping is one to many; i.e., several fragments correspond to one global relation, but only one global relation corresponds to one fragment.

Fragments are logical portions of global relations which are physically located at one or several sites of the network. The **allocation schema** defines at which site(s) a fragment is located. The type of mapping defined in the allocation schema determines whether the distributed database is redundant or nonredundant: in the former case the mapping is one to many, while in the latter case the mapping is one to one. All the fragments which correspond to the same global relation R and are located at the same site j constitute the physical image of global relation R at site j . There is therefore a one to one mapping between a physical image and a pair \langle global relation, site \rangle ; physical images can be indicated by a global relation name and a site index.

An example of the relationship between the object types is given in Fig. 3. A global relation R is split into four fragments R_1, R_2, R_3 and R_4 . These four fragments are allocated redundantly at the three sites of a computer network, thus building three physical images R^1, R^2 and R^3 . (R_i indicates the i th fragment of global relation R , and R^j indicates the physical image of the global relation R at site j).

Here we note that two physical images can be identical, which is a physical image of another physical image.

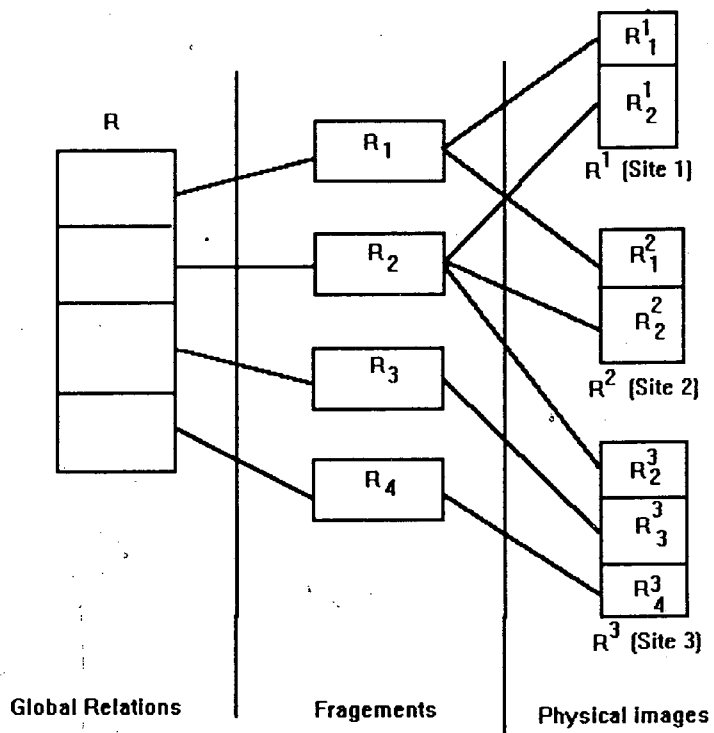


Fig. 3: Fragments and Physical Images For a Global Relation

In the architecture the three levels are site independent; so they do not depend on the data model of the local DBMSs. At a lower level, it is necessary to map the physical images to the objects which are manipulated by the local DBMSs. This mapping is called a **local mapping schema** and depends on the type of local DBMS; therefore in a heterogenous system we have different types of local mapping at different sites.

This architecture provides a very general conceptual framework for understanding distributed databases. The three most important objectives which motivate the features of this architecture are the separation of data fragmentation and allocation, the control of redundancy, and the independence from local DBMSs.

1. Separating the concept of data fragmentation from the concept of data allocation. This separation allows us to distinguish two different levels of distribution transparency, namely **fragmentation transparency** and **location transparency**. Fragmentation transparency is the highest degree of transparency and consists of the fact that the user or application programmer works on global relations. Location transparency is a lower degree of transparency and requires the user or application programmer to work on fragments instead of global relations; however, they do not know where the fragments are located. The separation between the concept of fragmentation and allocation is very convenient in distributed database design, because the determination of relevant portions of the data is thus distinguished from the problem of optimal allocation.

2. Explicit control of redundancy. The reference architecture provides explicit control of redundancy at the fragment level. For example Fig. 3 shows two physical images R^2 and R^3 are overlapping; i.e., they contain common data.

3. Independence from local DBMSs. This feature, called **local mapping transparency**, allows us to study several problems of distributed database management without having to take into account the specific data models of local DBMSs.

Another type of transparency which is strictly related to location transparency is **replication transparency**. Replication transparency means that the user is unaware of the replication of fragments. Clearly, replication transparency is implied by location transparency; however, in certain cases it is possible that the user has no location transparency but has replication transparency (thus, only one particular copy is used and the system makes appropriate actions on the other copies).

3.2. Functions of DDBMS

Now we will define the desired functions of a distributed DBMS by a collection of transparencies:

1. Location Transparency:

A major objective of distributed database system is to provide ease of access to data for users at many different locations. To meet this objective, the distributed database system must provide what is called **location transparency**. Location transparency means that a user requesting data need not know at which site these data are located. Any request to retrieve or update data at a nonlocal site is automatically forwarded by the system to that site. So, ideally the user is unaware of the distribution of data and all data in the network appears as a single logical database. Thus it means that a user can submit a query that accesses distributed objects without having to know where the objects are.

2. Performance Transparency:

Performance transparency means that a distributed query optimizer has been constructed to find a heuristically optimized plan to execute any distributed command. Obviously, if one has 1,000,000 objects in New York and 10 objects in Berkeley, one wants to perform the join by moving the 10 objects to New York and not vice versa. Performance transparency loosely means that a query can be submitted from one node in a distributed DBMS and it will run with comparable performance.

3. Copy Transparency:

Copy transparency means that the system supports the optional existence of multiple copies of database objects. Hence, if a site is down, users can still access database objects by obtaining one of the copies from another site.

4. Transaction Transparency:

Transaction transparency means that a user can run an arbitrary transaction that updates data at any no. of sites, and the transaction behaves exactly like the transaction either commits or aborts and no intermediate states are possible.

5. Fragment Transparency:

Fragment transparency means that the distributed DBMS allows a user to cut up a class (relation) into multiple pieces and place these pieces at multiple sites according to distribution criteria. For example, the following distribution criteria might control the placement of tuples from EMP relation

EMP where dept = "shoe" at Berkley

EMP where dept != "shoe" at New York

In this way the tuples of a relation can be distributed and user can access the EMP relation unaware of this distribution.

6. Schema Change Transparency:

Schema change transparency means that a user who adds or deletes a database object from a distributed database need make the change only one (to the distributed dictionary) and does not need to change the catalog at all sites that participate in the distributed databases.

7. Local DBMS Transparency:

Lastly, local DBMS transparency requires that the distributed database system be able to provide its services without regard for what local database systems are actually managing local data.

But it is extremely difficult or impossible to construct any distributed optimizer which support all of the above transparencies. Together with the transparencies the distributed optimizer should deal the following problems:

- Load balance
- Machine speed differences
- Network nonuniformity
- Administrative constraints
- Cost constraints
- Space constraints.

3.3. Distribution Transparency for Read-Only Applications: An Example

We consider a simple application, called SUPINQUIRY, which consists in accepting a supplier number from a terminal, finding the corresponding supplier name, and displaying it at the terminal. We will analyze how this application sees the database at decreasing levels of distribution transparency.

An SQL statement in this application defines a required database access primitive; it can be interpreted as the invocation of a procedure which receives some input parameters from the Pascal-like program, accesses the database, and returns the result as an output parameter. In order to indicate the input and output parameters for these database access procedures, the Pascal variables which are used as parameters in the SQL statement are prefixed with a "\$" symbol. The input parameters of an SQL query appear in the where-clause, while the output parameters appear in the select-clause. For instance, the query

```
select NAME into $NAME
from SUPPLIER
where SNUM = $$SNUM
```

can be considered as a procedure which receives the variable \$\$SNUM as an input parameter, selects the name of the supplier who has the current value of \$\$SNUM as supplier number, and returns this name to the Pascal-like program in the variable \$NAME.

In some cases, the application has also to communicate with the distributed DBMS in order to exchange some control information. The Pascal variable which are used for this type of communication are prefixed with the "#" symbol.

Level 1: Fragmentation Transparency The way in which the application accesses the database if the DDBMS provides fragmentation transparency is shown in Fig. 4a. First the application accepts a supplier number from the terminal; then it accesses the database. The whole SQL statement represents a single distributed databases access primitive, which receives the variable \$SUPNUM as input parameter and returns the variable \$NAME as output parameter. The DDBMS interprets this primitive by accessing the databases at any one of three sites in a way which is completely determined by the system. From the viewpoint of distribution transparency, the application refers to the global relation name SUPPLIER, completely ignoring the fact that the database is distributed. In this way, the application is completely immune to any change which is applied to all schemata which are below the global schema in our reference architecture.

Level 2: Location Transparency If the DDBMS provides location transparency but not fragmentation transparency, the same application can be written as shown in Fig. 4b. The request for the supplier with the given number is first issued referring to fragment *SUPPLIER*₁, and if the DDBMS returns a negative answer in the control variable #FOUND, a similar request is issued with respect to fragment *SUPPLIER*₂. At this point, this naive implementation assumes that the supplier has been found and displays the result. Several variations are possible, for instance, issuing both requests in parallel in order to exploit the parallelism of distributed system; however, this does not change the distribution transparency characteristics. This application is independent from changes in the allocation schema, but not from changes in the fragmentation schema, because the fragmentation structure is incorporated in the application. However, location transparency is very useful because it allows the applications to ignore which copies



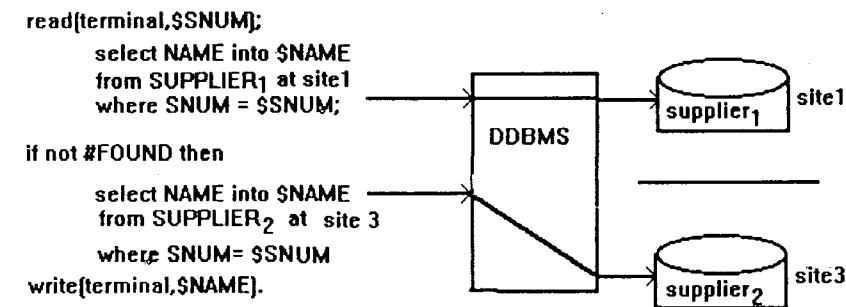
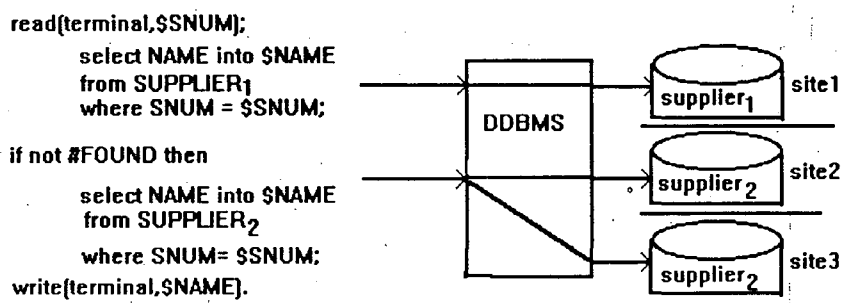
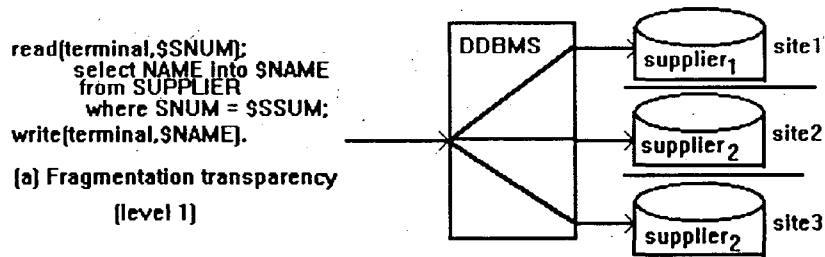


Fig. 4. The read-only Applications SUPINQUIRY at different levels of Distribution Transparency

exit of each fragment, therefore allowing copies to be moved from one site to another and allowing the creation of new copies without affecting the applications.

When location transparency is provided without fragmentation, it is very efficient to write applications which take explicit advantage of knowing the fragmentation structure. For example, the same application of Fig. 4b can be written in the following way:

```
SUPINQUIRY:

    read (terminal, $SNUM);

    read (terminal, $CITY);

    case $CITY of

        "SF": Select NAME into $NAME
              from SUPPLIER1
              where SNUM = $SNUM;

        "LA" : Select NAME into $NAME
              from SUPPLIER2
              where SNUM = $SNUM

    end;

    write (terminal, $NAME).
```

Level 3: Local Mapping Transparency At this level we assume that the application still refers to objects using names which are independent from the individual local systems; however, it has to specify at which site the objects reside. The site names are indicated in the SQL statements by adding an "at" clause to the form "from" clause. Clearly in this case each database access uses site-independent fragment names. If the mapping were not provided, the application would incorporate directly the filenames which are used by the local systems.

The most important aspect of local mapping transparency is not this name-mapping between fragment names and local filenames, but the mapping of the primitives used by the application program into the primitives used by the local DBMS. Therefore the local mapping transparency is an important feature in a

heterogenous DDBMS. Suppose that the local DBMS at site 1 is IMS and the local DBMS at site 3 is a Codasyl. In order to provide local mapping transparency, the DDBMS has to perform the database access primitives issued by the application into corresponding IMS and Codasyl programs. Similar translation problems can be found in a homogenous DDBMS, because even in homogenous systems there is a difference between the primitives which are useful at the global level and the primitives of the local DBMS.

4 USE OF TRANSPARENCY IN VIEWS OF DISTRIBUTED DATABASES

4.1. Introduction

The structure of data to be stored by a Data Base Management System (DBMS) is usually decided by a database administrator. Individual users and applications are generally interested in only a subset of the data stored in the database. Often, they wish to see this subset structured in a way which reflects their particular needs. Since it is not generally possible to structure a database so as to please all of its users, some mechanism is needed whereby each user can view the data according to his own requirements. The representation of the data structure as seen by a user is often referred to as an external schema; the view mechanism is a means by which a DBMS can support various external schemas.

Besides providing users with tailored views of the data, the view mechanism contributes to :

- Data independence: giving applications a logical view of data, thereby isolating them from data reorganization.
- Data isolation: giving the application exactly that subset of data it needs, thereby minimizing error propagation.

In a relational DBMS, a view is defined as a "virtual table" derived by a specific query on one or more base tables. The relational operations join, restrict and project as well as statistical summaries of tables may be used to define a view. Access rights may be granted and revoked on views just as through they were ordinary tables. This allows users to selectively share data, preventing unauthorised users from reading sensitive information.

4.2. Views in Distributed Database Systems

Among numerous goals of a distributed DBMS, two have been recognized as key objectives:

- site autonomy and
- data distribution transparency.

Site Autonomy

Site autonomy means that each site can operate on its own data as a stand-alone, single-site DBMS, and that each site retains local control of its own data, even if the site participates in the execution of a distributed query. This guarantees better resiliency of failures of sites and communication lines, since there are no centralized functions or services, such as a global dictionary or centralized deadlock detector. Further, each site performs all operations on its own local data, including authorization checking and database accesses and updates.

Site autonomy in multidatabases: A key aspect of multidatabases is that each local DBMS retains complete control over local data and processing. This is referred to as site autonomy. Each site independently determines what information it will share with the global system, what global requests it will service, when it will join the multidatabases, and when it will stop participating in the multidatabases. The DBMS itself is not modified by joining the multidatabases. Global changes, such as addition and deletion of other sites, or global optimization of data structures and processing methods, do not have any effect on the local DBMS. Local DBAs are free to optimize local data structures, access paths, and query-processing methods to satisfy local user requirements rather than global requirements. Since the global system interfaces with the local DBMS at the user level, the local DBMS has as much control over the global system as it does over local users.

The multidatabase approach of preserving site autonomy may be desirable for a number of reasons. Some local databases may have critical roles in an

organization, and it may be impossible from an economic standpoint to change these systems. Site autonomy means the local DBMS can add global access without changing existing local function. Another economic factor is that an organization may have significant capital invested in existing hardware, software, and user training. All of this investment is preserved when joining a multidatabases since existing local applications can continue operating unchanged. Site autonomy can also act as a security measure because the local DBMS has full control over who accesses local resources through the multi database interface and what processing access will be allowed. In particular, a site can protect information by not including it in the local schema that is shared with the global system. An organization's requirements for global access may be minimal or sporadic. Site autonomy allows the local DBMS to join and quit the multi database with minimal local impact.

Despite the desirable aspects of site autonomy, it places a large burden on global DBAs. Each site has independent local requirements and makes independent local optimization to satisfy those requirements. Because of this independence and the possibly large number of participating sites, global requirements and desirable global optimization are likely to conflict with local ones. The global DBA must work around these conflicts in initial global system design and ongoing global maintenance. Global performance suffers relative to a tightly coupled distributed data base because of the lack of global control over local resources. Because of the heterogeneity of local DBMSs, the global system may have to dedicate global resources to compensate for any missing local function or information. Some of these problems may be alleviated to a degree if the local DBAs agree to cooperate and conform to some global standards. Site autonomy ensures that this cooperation is not enforced by the system, but organizational policies can be used to force cooperation.

Distribution Transparency: The second objective, distribution transparency, means that users are shielded from the physical distribution and redundancy of data and are

able to interact with the distributed system as easily as with a conventional centralized one. This ensures logical independence of applications.

Any extension of views to a DDBMS must preserve the appearance of views as virtual tables. This presents many problems due to the fact views definition site and/or using other views defined at remote sites. Views defined using non-local objects are themselves distributed objects, which requires that the operations of creating, dropping and using a view be distributed operations.

The issue of authorization on views also has major implications for the implementation. When a view is used in a query, view composition must take place in order to derive an execution strategy for the query. If views are not objects of authorization, this composition can take place at any site. If views are objects of authorization, site autonomy consideration require that the view definition site maintain control over the materialization of the view. In particular, view composition must take place at the view definition site. If view composition is allowed to occur at any site, a malicious site could prevent the view definition by dropping restrictions or projections in the view definition.

Forcing view composition at the view definition site can have negative effects on performance. If the definition site of a view is not the site at which a query using the view is submitted (query master site), then it may not be possible for a single site to produce a complete execution strategy for the query, because of the view definition site. Further, in systems which separate planning a query from its execution, the execution strategy must include accessing the view definition site to check that the user is still authorized to use the view. This must be done at execution time even if no tables referenced in the fully composed query are stored at the view definition site.

For these, the execution strategy for a query referencing a remotely defined view is unlikely to be optimal, and a performance penalty may be incurred. If the views were not objects of authorization the query site could produce a complete

execution strategy, which would not require accessing the view definition site unless some table were actually stored there.

Hence, two types of views are recognized. Shorthand views provide the data hiding, data conversions, typing elimination and renaming functions associated with views, but are not objects of authorization. The user posing queries against a shorthand view must be authorized to access the objects referenced by the view. Protection views provide the same semantics as shorthand views and in addition are objects referenced by the protection view belong to the view and the user of the view only needs the privilege to use the protection view. Queries referencing remotely defined shorthand views will in general execute more efficiently than identical queries with shorthand views replaced by protection views.

4.3. Distributed View Management

Views are defined in terms of queries which may reference local and non-local tables and views. Queries may be imbedded in programs, which are precompiled by DDBMS's. The result of precompilation is a set of access modules, defining the execution plan for query, which are stored in the distributed database. Precompilation also creates dependencies for the program on the tables and views referenced in the program must be invalidated. At execution time of a program, the system checks whether the program is valid. If the program is still valid, the system loads it and executes the necessary access modules. If the program has been invalidated, the system may try to recompile the program; if recompilation succeeds, the program is executed, otherwise an error is reported.

Since distribution transparency requires that the system have the same behaviour with respect to users as a centralized DBMS, a correct implementation of views in DDBMS must ensure that views are dependent on the objects they reference, and that the programs referencing views are independent on those views. These requirements ensure a consistent usage of views by users.

In addition, site autonomy requires that each site be able to perform any action on local (non-distributed) objects, such as dropping local tables or purely local views, without notifying any other site. In particular, it should not be necessary to contact other sites at which programs or views referencing the local objects are stored or defined.

These two requirements suggest that dependency recording should be distributed among the sites of those objects on which the view (or program) depends. In other words, view or program dependencies on remote tables are stored. This allows local invalidation of distributed programs or remotely defined views if a local table is dropped or changed.

Actually the situation is somewhat more complicated. Programs and views may depend on remotely defined protection and shorthand views, as well as remotely stored tables. Dependencies on protection views are recorded at the view definition site. In fact this site will be accessed at execution time, since the view is materialized at that site.

If the dependency of a program on a shorthand view is recorded at the view definition site, and that view is later dropped, it may not be possible to invalidate the program. This will happen if no table referenced by either program or view is stored at the view definition site. In this situation, the program has no need to access the view definition site at execution time and hence it will not discover that the view has been dropped.

To invalidate a program in these circumstances, dependencies on shorthand views are recorded at other sites, chosen in such a way that any program referencing the view must access these sites at execution time. In fact, these sites are the sites at which tables referenced by the view are stored.

An obvious consequence of these distributed dependencies is that view definition and view drop are distributed operations. At view definition time, the dependency of a view on remote tables must be recorded at the table store sites. When dropping a view, remote sites storing view dependencies must be accessed in

order to delete these dependencies and at the same time, invalidate programs and views depending upon the dropped view. The remote sites are only accessed when the view is a distribute object. Local views are still dropped locally.

5 PROS AND CONS OF TRANSPARENCY

Distributed database software offers transparent access to data--- no matter where in the network the data is located, an authorized program can access the data as though it is local. Transparency has been the goal of distributed database systems for over a decade--- it is at the core of next-generation distributed database systems. The real virtue of transparency is its ability to support geographically centralized clusters of computers.

The IBM's CICS/ISC and Tandem's Encompass have offered transparency. Although these two distributed systems are very popular, their ability to transparency access remote data is not used much. In fact, design experts of both vendors recommend against transparent remote access to data; instead, they recommend a requester-server design (sometimes called remote procedure call) in which requests for remote data are sent to a server which accesses its local data to service the request. Ironically, CICS and Encompass initially offered only transparent distributed data.

Manageability is the major problem for the geographically distributed applications. This is the reason that the old-timers, who had the facility of transparency, are skeptical about using transparency for geographically distributed applications, while the newcomers are enthusiastic about the transparency in these geographically dispersed applications, and so we can say that the geographically distributed applications are a nightmare to operate. By definition they are a large and complex system involving hundreds if not thousands of people. Communication among computers, administrators and operators in such a system is slow, unreliable, and expensive (SAU). When confronted with a "real" distributed system, one with SUE communication, successful application designers fall back on the requester-

server model. Such designs minimize communication among sites and maximize modularity and local autonomy for each site.

Now the question arises that whether the distributed database have an important application: they allow modular growth-- the ability to grow a system by adding hardware modules to a cluster instead of growing by trading up to a bigger, more expensive box. It is not enough to have a "cluster" hardware architecture-- distributed system software is needed to allow modular growth. Clusters avoid all the hard problems of geographically distribution:

-- Clusters have high-bandwidth, low-latency, reliable and cheap communication among all processors in the cluster. All data in the cluster is "close" since access times are dominated by disk access time. Geographically distributed system must deal with slow, unreliable, expensive communications via public networks. In geographically distributed systems message delays dominate access times.

-- A cluster can be administered and operated by a single group with face-to-face contact and with similar organizational goals. "Real" distributed systems involve multiple sites, each site having its own administration. Personal communication and relations among sites are via slow, unreliable, expensive mail and telephone -- SUE again.

5.1. A Case Against Transparent Access to Distributed Data

Distributed databases offers transparent access to data. Beyond the authorization mechanism, there is no control over what the program does to the data. It can delete all the data, zero it, or insert random new data. In addition, no comprehensible audit trail is kept telling who did what to the data. This interface is convenient for programmers, but it is a real problem for application designers and administrators.

The simplest way to explain the negative aspects of a distributed database is to compare refrigerators to grocery stores. My refrigerator operates like a distributed database. Anyone with a key to my house is welcome to take things from the

refrigerator or put them in. There is a rule that whoever takes the last beer should get more at the grocery store.

Requester-server designs provide an administrative mechanism. They provide defined, enforceable, auditable interfaces which control access to an organization's data. Rather than publishing its database design and providing transparent access to it, an organization publishes the CALL and RETURN messages of its server procedures. These servers perform requests according to the procedures specified by the site owner. They are site's standard operating procedures. Requesters send messages to servers which in turn execute these procedures and enforce the store's operating procedures.

Request-server designs are more modular than distributed databases. A site can change its database design and operating procedures without impacting any requesters. This gives each site considerable local autonomy. The only thing a site cannot easily change are the request and reply message formats. In the parlance of programming languages, distributed databases offer transparent types, servers offer opaque types, sometimes called abstract types or encapsulated types.

Request-server designs are more efficient. They send fewer and shorter messages. Considering the example of adding an invoice to a remote node's database. A distributed database implementation would send an update to the account file, insert a record in the invoice file, and then insert several records in the invoice-detail file. This would add up to a dozen or more messages. A requester-server design would send a single message to a server. The server would then perform the updates as local operations. If the communication net is SUE then sending only message is a big savings over multi-message designs.

Thus the disadvantages of the applications with transparent access to geographically distributed databases as compared to a requester-server design are:

- Poor manageability,
- Poor modularity and,
- Poor message performance.

The lunatic fringe of distributed database promise transparent access to heterogeneous database. These folks promise to hide all the nasties of networking, security, performance, and semantics under the veil of transparency. But the prospect of getting people who cannot agree on how to represent the letter "A" to agree to share their raw data is far fetched. Heterogeneous systems are a very good argument for requesters and servers. The systems need only agree on a network protocol and a requester-server interfaces. Still a little far fetched unless a standard network and requester-server model emerges.

5.2. Manageability of a Distributed Database

If we have to design and manage a distributed applications in the real world then we can not use transparent access to geographically remote data, because a distributed system is a big and complex thing and it needs to change and grow over time. We may want to add nodes, move data about, redesign the database, change the format or meaning of certain data items, and do other things which are likely to invalidate some programs using the data.

5.3. The Case For Transparent Access to Cluster Data

The transparent access is very convenient for programmers— it makes it easy to bring up distributed applications. Coding requesters-servers and making an application modular is extra work. Clustering is the real application of transparent access to distributed data. It offers both the customer and the vendor significant advantages. The customer can buy just what he needs and grow in small increments as he needs more. The vendors has two advantages. First it needs design and support only a very few module types. In addition, it can build systems which are exceed the power of the non-clustered vendors.

The arguments against geographically distributed database do not apply to clustered systems. A cluster and its operators are typically in a single room. SUE is

not a problem. The people have face-to-face contact and the computers have duplexed, high-speed buses among them.

A cluster is like a centralised system, so it can be managed as one. For small clusters the local autonomy derived from modularity may be moot. A distributed database server cluster applications nicely, allowing data to be partitioned among any disks in the cluster and allowing servers to run on any cpus in the cluster. Because the intra-cluster communication is fast and cheap, the cost of distributing data in the cluster is negligible.

6

R* : AN EXAMPLE OF LOCATION TRANSPARENCY

R* is an operational, experimental, distributed databases management system which supports the structured query language (SQL). The most important object of R* is to provide site autonomy. Site autonomy is achieved when each site is able both to control accesses from other sites to its own data and to manipulate its data without being conditioned by any other site. While the first goal is completely achieved by R*, the second goal is partially achieved, since a loss of site autonomy cannot be avoided during the 2-phase-commitment of transactions.

The site autonomy also requires that the system be able to grow incrementally and to operate continuously, with new sites joining to existing ones, without requiring existing sites to agree with joining sites on global data structures or definitions.

Another important issue in R* is location transparency (the user is not aware of the actual location of data); thus, from the programmer's viewpoint, the use of R* is essentially equivalent to the use of a centralized system. The relevant extensions to the SQL language due to distribution are:

1. Naming, an R* systemwide name includes for each object the specification of

`<creator>@<creator-site>.<object>@<birth-site>`

This naming scheme includes the birth site of each object, which is a static property, but does not include the actual location of the object, which can change from time to time.

2. The possibility of moving objects between sites.

R* is composed of three components: a local database management system, a data communication component which provides message transmission, and a transaction manager which coordinates the implementation of multiple transactions.

The local database management is further divided into two components: a storage system, concerned with the storage and retrieval of data, a database language processor, which translates high-level SQL statements into operations on the storage system. The storage system used in the R* project, called RSS*, is based on the research storage system of System R.

R* sites communicate via the Inter System Communication (ISC) facility of CICS. Each R* site runs in a CICS address space, and CICS handles terminal I/O and message communication. The communication is assumed to be unreliable (i.e., there is no guarantee that a transmitted message will be eventually delivered), but it is assumed that delivered messages are correct, not replicated, and received in the same order as they were sent.

An application program makes all database access requests to the R* system at its local sites. All intersite communications are between R* systems at different locations, since R*, rather than an application program, is responsible for locating distributed data. Thus, in the R* environment there is no need for remote application programs.

6.1. Problems Solved by R* System

In a centralized database management system each database instance is located at a single geographical site. These centralized systems simplify the job of sharing data by providing a high-level database language (SQL), unit-of-work (i.e., transaction) management, concurrency management, authorization services and more. But what will happen to data sharing when an enterprise has more than one of these centralized DBMSs? For these applications that access only local data, there is no data sharing problem. However, for applications that need to access data at one or more remote databases, there is a data sharing problem and current centralized DBMSs do little help.

Fig. 5 illustrates this problem. Assume that widget salespeople in New York need to access both sales information located in their local database and inventory information located at the manufacturing plant in Los Angeles. They need this information in order to produce a report which predicts their ability to supply future demand for widget in the New York area. There are several ways for the salespeople in the New York to access the data located in the Los angeles: 1) They could telephone someone at Los Angeles and ask them to query the database and describe the results; 2) They could connect a terminal located in New York to application programs running against the Los Angeles database. 3) They could extract a data file containing the relevant inventory information from the Los Angeles database and transmit it to New York. Regardless of which these methods is used, the inventory data obtained from Los Angeles would then have to be combined with the sales data from New York by the application program or the salespeople themselves to produce the desired report. There are numerous variations on how one might overcome the difficulties created by the fact that some of the required data is stored in a remote database; but each variation adds complexity for either the application designer or end-user that not be necessary if all the data were located in a single DBMS.

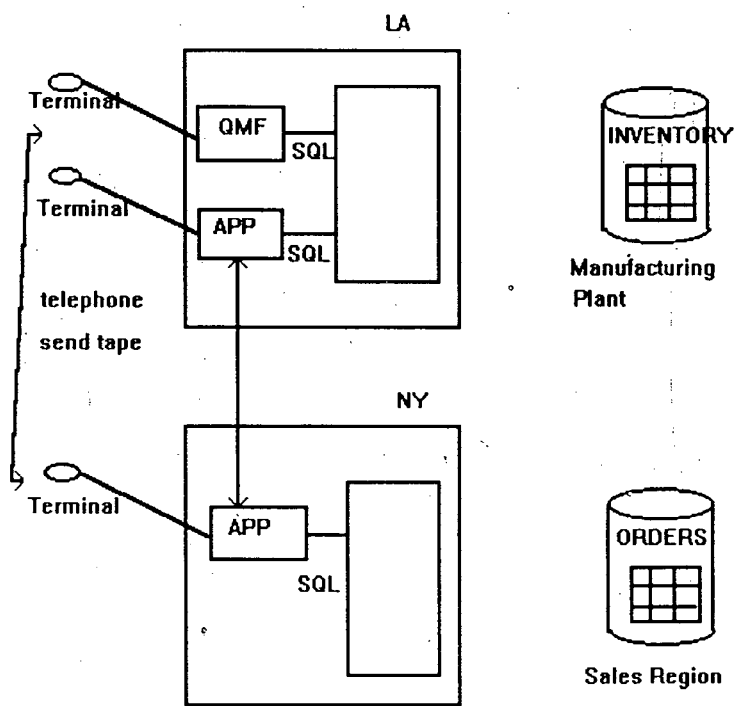


Fig. 5: Application Design is Difficult

A distributed DBMS, such as R*, simplifies this data sharing problem permitting application programs to access data objects stored in a network of interconnected database sites by simply issuing database requests. Assume for a moment that R* is installed at both Los Angeles and New York. Assume also that the custodians of the Los Angeles database have entered appropriate SQL commands to authorize access to portions of the inventory tables to the salespeople at New York. An application program could then be written that contains SQL statement that reference the sales information tables located in the New York database and inventory information tables located in the Los Angeles database. The salespeople could then run the application at New York and generate the report.

6.2. View Management in R*

A view in R* is a non-materialized virtual relation defined by an SQL statement. It is defined in terms of one or more tables or previously defined views and during processing a view is materialized from its component objects. Views can be used as shorthand notation for reducing the amount of typing required when frequently executing queries, or they can be used as a protection mechanism for hiding rows or columns in underlying tables from the user of the view. Thus we can say that views satisfy the property of location transparency.

In R* view component objects may be at different sites. Therefore to provide data protection between sites a protection view is materialized only at the site owning the view. This scheme prevents sending sensitive data to a node where no user is authorized to see it. Therefore during compilation the master site generates a plan for processing the view as if it were a physical table and sends the plan and SQL statements to the apprentices where the view will be processed. An apprentice site may itself decompose a view component in terms of other views on tables at yet other sites. In that case the apprentice acts as a master to other apprentices and must generate a plan and send subplans to its apprentices sites. The plan

distribution progresses in this way until all views are resolved and may even back to a site that has already participated in the compilation.

6.3. R* Objectives

R* was designed to satisfy three design objectives: it should 1) be easy to use, 2) be compatible of autonomous operation, and 3) provide good performance. To satisfy these objectives, many difficult design choices were addressed such as query decomposition, optimal site and access path selection, transaction consistency, locking and detection, recovery and resilience with respect to site failure.

The first objective is ease to use. R* is easy to use because application program developers and end-users can use SQL to access data objects just as they do today in SQL/DS. The reason that R* users can use SQL is that R* supports the notion of **data location transparency**. The database worries about where an object is currently stored, not the application programmer or end-user. In fact R* a single application program can be written which executes at many different database sites in the network. At each site, a specially tailored access plan is developed by the database to optimize access to the database objects defined in the application, giving its execution location and the current locations of the database objects that it references.

The second R* objective is **site autonomy**. One way to characterize a distributed system is the degree to which a single node in the system can operate when the connection to other nodes fail. In R*, this ability to operate autonomously is considered vital. Sites should also administered independently. They should be able to add new user to their system, switch to new releases etc., without consulting any central authority. To achieve this degree of autonomy, each database site must be entirely selfsufficient, it cannot rely on any form of central service such as a central name server or a central point for detecting deadlocks. Autonomy also implies that individual sites do not have global knowledge about what operations taking place at

other sites and the individual sites will continue database processing despite site and/or communication failures.

Third and final R* objective is **good performance**. There are many things in R* that were done to promote good performance. One is that statements are compiled into access modules. During compilation, database object names are resolved, access paths are determined, authorization rights are validated, and access plans are distributed to all involved database sites. This early binding makes it unnecessary to repeat many of these time consuming operations each time the application program is actually executed. The other thing that leads to good performance is that R* enables user to place copies in of data in their local system to avoid the overhead of using the communication system when their applications are executed. This concept is called **locality of reference** and is vital to good performance in any distributed system which utilizes relatively slow and unreliable communication devices. A user can invoke new MIGRATE TABLE statement to physically move table from a remote database site to the local database site. Alternatively, a user can create a snapshot of data. A snapshot is a stored database object, like a table, but it is for read-only access and it periodically refreshed from the base objects in its definition. The definition of a snapshot is in the form of an arbitrary SQL SELECT query.

Thus the R* prototype makes it possible to build a distributed SQL system that is easy to use, can run autonomously and has good performance. It appears likely that workstation databases can be incorporated into the R* design and can thereby increase the degree of potential data sharing. Instead of just a few dozen large hosts sharing data, there may be thousands of workstations connected together or connected to a back-bone host database network. The level of potential data sharing in such a network is mind-boggling. Much research needs to be done to understand the various design alternatives and to understand how to configure and administer such distributed database networks.

Thus the R* achitecture supports several kinds of data distribution. Attention has been given to efficient execution of programs by the compilation and optimization of users queries and SQL programs. It emphasizes on site autonomy. At the same time, R* provides transparent remote data definition and manipulation facilities, distributed transaction management, and distributed concurrency control which should simplify data sharing for both ad hoc query users and application programmers. Existing single site SQL programs and queries can be run against distributed data without modification in an R* environment and programmers can continue to develop programs without having to worry about network issues.

7 CONCLUSION

For geographically distributed databases, to achieve the transparency at various levels together with the distribution of data, there are four distributed data allocation strategies which are distinct but interdependent: data fragmentation, data placement, data replication and dynamic data allocation. Among these four main emphasize is given on data fragmentation which provides advantages of data locality and system environment. We can divide the database in mainly two ways i.e. horizontally or vertically. In both cases we can get a different level of transparency.

Distribution transparency provides the independence of programs from the distribution of the databases. Different levels of distribution transparency can be provided by a DDBMS; at each level, different aspects of the real distribution of data are hidden from the application programmers. At the highest level, called fragmentation transparency, a modification of data distribution does not require rewriting programs. Providing distribution transparency for update applications is because update application attributes; in this case, a rather complex restructuring of affected data might be required.

Thus we can see that distributed database software offers transparent access to data irrespective of the location of the data in the network, an authorized program can access the data as though it is local. So the transparency in a geographically distributed system is unmanageable and has technical drawbacks. The real virtue of transparency is its ability to support geographically centralized clusters of computers.

BIBLIOGRAPHY

- [1] Apers P.M.G., "Data Allocation in Distributed Database Systems," ACM Transactions on Database Systems, Vol. 13, No. 3, Sept. 1988, pp. 263--304.
- [2] Bertino E., Haas M.J. and Lindsay B.G., "View Management in Distributed Database Systems," Proc. 1983 VLDB, pp. 376--378, Oct. 1983.
- [3] Ceri S. and Pelagatti G., "Distributed Databases: Principle and Systems," McGraw-Hill, New York, 1984.
- [4] Ceri S. and Navathe S.B., "A Methodology for the Distribution Design of the Databases," Proc. Compcn 83, San Francisco, march 1983.
- [5] Ceri S., Negri M. and Pelagatti G., "Horizontal Data Partitioning in Database Design," In Proceedings of ACM-SIGMOD Conference, ACM, New York, 1982.
- [6] Gray J., "Transparency in its Place---The Case Against Transparent Access to Geographically Distributed Data," UNIX REVIEW, Vol. 5, No. 2, May 1987.
- [7] Hevner A.R. and Rao A., "Distributed Data Allocation Strategies," Information Systems Department College of Business and Management, University of Maryland, College Park, Maryland 20742.
- [8] Hurson A.R. and Bright M.W., "multidatabase Systems: An Advanced Concept in Handling Distributed Data," Advances in Computers, Vol. 32, pp. 149--197.
- [9] March S.T. and Rho S., "Allocating Data and Operations to Nodes in Distributed Database Design," IEEE Transactions on knowledge and Data Engineering, vol. 7, No. 2, pp. 305--320, Apr. 1995.
- [10] Navathe S.B., Ceri S., Wiederhold G. and Don J., "Vertical Partitioning for Physical and Distyribution Design of Databases," Report No. STAN-CS-82-957, Stanford University, Stanford, 1982.

- [11]. Navathe S., Ceri S., Wiedershold G. and Don J., "Vertical Partitioning Algorithms for Database Design," ACM Transactions on Database Systems, Vol. 9, No. 4, pp. 680--710, Dec. 1984.
- [12]. Parent C., "Integrity in Distributed Databases," Proc. AICA 77, Pisa, 1977.
- [13]. William R., Daniels D., Haas L., Lapis G., Lindsay B., Ng P., Obermarck R., Selinger P., Walker A., Wilms P. and Yost R., "R*: An Overview of the Architecture," Improvings ed., academic Press, Inc, New York, pp. 1--27, 1982.
- [14]. Yost R.A. and Haas L.M., "R*: A Distributed Data Sharing System," Computer Science, IBM Research report RJ4676 (49844), Apr. 1985.