SISU
analys
Nr. 6

An Introduction to Distributed Database Systems

by

Moira Norrie

Redaktör: Benkt Wangler
SISU - Svenska Institutet för Systemutveckling
Box 1250, 163 13 SPÅNGA, 08 - 750 7500, Kistagången 26, KISTA

Innehållsförteckning

Förord

Detta nummer av SISU Analys redogör för forskningsområdet distribuerade databassystem. Eftersom området är nytt och svårt, finns ännu så länge endast prototypsystem i forskningsmiljö realiserade. Det är därför i första hand allmänna problemställningar och teorier som presenteras. Några viktiga forskningssystem nämns dock kort.

Ett distribuerat databassystem är ett system som medger enhetlig tillgång till data som hanteras av ett antal olika databashanteringssystem. Dessa tänkes normalt vara fördelade över flera datorsystem, sammankopplade i ett nätverk, därav benämningen distribuerad. Om de olika databashanteringssystemen tillämpar samma typ av datamodell, t.ex. relationsmodellen, säges systemet vara homogent, i annat fall heterogent.

I fortsättningen kallas de olika datorsystemen i nätverket för noder. Dessa kan t.ex. vara lokaliserade till olika delar av ett företag. Det praktiska och säkerhetsmässiga värdet av att på detta sätt sprida data och inflytande över dessa till de platser där det hör hemma, är uppenbart.

Inledningsvis redogörs för begrepp och terminologi inom området, liksom för olika kategorier av distribuerade databassystem. Därefter följer en genomgång av vart och ett av problemområdena

- datadistribution
- bearbetning av utsökningskrav
- transaktionshantering
- globala datamodeller och språk
- schemaintegration.

Datadistribution har att göra med hur data fördelas över de olika noderna i nätverket. Här finns väsentligen två vägar att gå. Den ena innebär att man har samma typer av uppgifter på varje nod, men på var och en enbart den del av den totala mängden data som rör den noden. Den andra och mer komplicerade metoden innebär, något förenklat, att man har olika slag av information (olika projektioner), men avseende samma objekt i verksamheten, på de olika noderna. Upprepning av information på flera noder kan av effektivitetsskäl ibland vara motiverad.

En viktigt problem vad gäller *bearbetning av utsökningskrav* är hur svar på frågor som kräver sammanställning av data från flera noder skall produceras så effektivt som möjligt. Lösningen på detta problem är att, utan att äventyra slutresultatet, ändra den ordning i vilken de olika operationer som ingår i utsökningen utföres, så att den totalt överförda datamängden minimeras.

Eftersom en *transaktion* kan initieras vid vilken nod som helst och beröra flera andra noder, är sekvenserings-, 'deadlock'- och 'recovery'-problematik ett angeläget forskningsområde. På grund av förekomsten av distribuerade transaktioner kommer här traditionella strategier delvis till korta och dessa behöver därför vidareutvecklas och förbättras.

Även för ett heterogent system är det lämpligt att, för kommunikation mellan noder, ha en gemensam *global datamodell* med tillhörande frågespråk. Relationsmodellen är här, på grund av sin flexibilitet, ett naturligt val.

Om man vill skapa en distribuerad databas utifrån ett antal existerande databaser, ställs man inför problemet att *integrera* dessas *lokala schemata*. Huvudproblemet är i detta fall att identifiera begrepp som är semantiskt ekvivalenta och att avgöra på vilket sätt de är detta. Exempelvis kan två databaser innehålla information om samma objekt i verksamheten, men på olika detaljeringsnivå eller med olika strukturell representation.

Det mesta av den forskning som hittills bedrivits har gällt homogena system, eftersom man där slipper problemen med att transformera mellan olika datamodeller och frågespråk. På senare tid har dock intresset för heterogena system ökat, eftersom det potentiella användningsområdet för sådana är större.

Benkt Wangler

An Introduction to Distributed Database Systems

Moira Norrie

September, 1987.

0. Introduction

The recent advances in computer networking in terms of both the technology and the standardization of protocols, has resulted in an increased interest in distributed database systems. Although still at the research stage, it is envisaged that such systems will be both feasible and cost-effective in the near future.

Here, we discuss the general requirements of distributed database systems and provide an overview of the techniques used to satisfy these requirements. We wish to emphasize that the term *distributed database system* is a general term that can be applied to many different kinds of systems, and that the requirements of any one system will depend on the intended use and environment of that system. Hence, any two such systems will share some, but not necessarily all, of the requirements discussed in this paper.

In section 2, we shall examine some of the categories of distributed database systems and consider their requirements. The later sections,will then introduce techniques to meet these requirements. Here, it is possible only to provide an overview of these techniques. For a detailed discussion of distributed database systems, one should turn to the text by Ceri and Pelagatti [3] - both for their detailed descriptions of the techniques and for its valuable bibliographic information.

First, we have a section on terminology to introduce terms and notation that will be used in our discussion, and, hopefully, provide sufficient background knowledge for those readers with little experience of database concepts.

1. Terminology

A *database* is a collection of data which is shared by a number of applications. Access to the database is controlled by a software system known as a *database management system* (DBMS).

Associated with a database will be a database *schema* which is a general decsription of the information represented in the database. The DBMS will utilise the schema in performing operations on the database.

Operations on the database are specified by means of a database language. In some systems, there are two kinds of database language - one for specifying the schema (Data Definition Language) and one for specifying operations on the database (Data Manipulation Language). Other systems treat data and metadata (the schema) uniformly and operate on both by means of a single database language.

The database language either may be embedded in a host programming language, or be used as a direct interactive language, in which case it is usually termed a *query language.* Ideally, a query language should allow the user to interact directly with the DBMS and query both the database and the schema. Some query languages only support retrieval operations while others support both update and retrieval operations.

The term *database system* will be used to refer to an entire application system i.e. a DBMS with a particular database schema, associated database and user queries/ application programs as illustrated in figure 1. A logical unit of work in a database system, whether a user query or part of an application program, is termed a *transaction.*

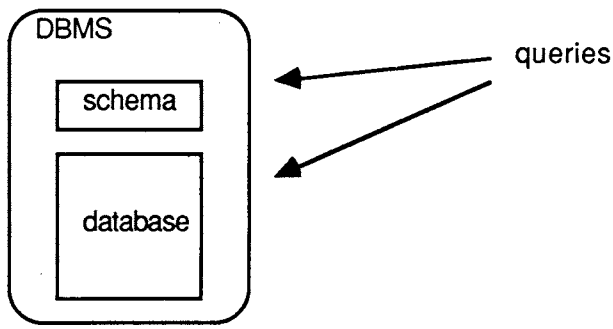


Figure 1. An Application System

Each DBMS is based on some *data model.* A data model may be thought of as a framework for the construction of schemata and associated databases in that it specifies a set of available structures and operations.

Many data models have been proposed, but the only one which will be mentioned in any detail in this paper is the relational data model. We shall therefore present a very simple overview of this data model which should provide sufficient information for the remainder of this paper. For further details of both the relational model and other data models, a general database text such as that by Ullman [7] is recommended.

A relation may be viewed as a table. Each row is referred to as a tuple. Each column contains values associated with some attribute. The number of columns defines the degree of a relation, and the number of rows its cardinality.

A relation is described by means of a relation scheme which gives the name of the relation and a list of its attributes. For example, a relation containing information on persons could have the following relation scheme:

PERSON (NAME, ADDRESS),

and a relation on this relation scheme could be:

PERSON

NAME	ADDRESS
Jones F	4 High St, Chester
Martin M	33 Main St, Falkirk
Smith W	76 Wood Lane, London
Thomas M	1 Mill Rd, Newtown
Watson J	15 North St, Glasgow

A relational database would consist of a number of relations and the schema for the database would include the set of corresponding relation schemes.

If an attribute is specified as a *key* for a relation, then the value of that attribute will uniquely specify a tuple of the relation. Thus, if NAME is a key attribute for PERSON, then only one tuple in the relation can have a particular name value.

There are many relational operators, but here we shall only consider those of projection, selection, union and join which will be denoted as follows:

$\pi_A(R)$	project tuples of R to attribute A
$\sigma_f(R)$	select those tuples of R that satisfy condition f
$R \cup S$	form the union of R and S
$R \bowtie S$	perform the join of R and S

Effectively, the selection operation selects a subset of the rows of a relation, while the project operation selects a subset of the columns and removes any duplicate rows that appear in the reduced relation.

If two relations, R and S, have the same scheme, then the union of these two relations would be a relation formed by taking the rows of R together with the rows of S - again removing any duplicates.

If two relations have a common attribute, then these two relations may be joined with respect to that attribute by forming a new relation which links together rows which have matching values for the common attribute. Thus, a join operation is a way of combining information held in two relations.

There are many different forms of join operation; however, our references to join

operations would apply to all forms of joins and we will therefore leave unspecified the "join condition" of the operations.

2. Categories of Distributed Database Systems

A survey of the literature on distributed database systems would reveal that while there is general agreement that certain kinds of system should be classified as *distributed database systems*, it is not very clear where the boundaries of this classification lie. So first we shall discuss some categories of system that might be classified as distributed database systems, and, further, consider the characteristics and requirements of such systems.

Our definition of a distributed database system is:

> *A distributed database system is a system which supports uniform access to data controlled by a number of database management systems.*

This definition is deliberately a very general one and, as we shall see, covers a wide range of systems: however, all of these systems could be considered as a restriction of a very general and flexible view of a distributed database system.

We shall discuss three general categories of systems covered by such a general definition and show that each category of system has its own intended uses, characteristics and hence requirements.

Let us begin by considering a system which provides

uniform access to a collection of different database systems.

We may assume that the information contents of the databases are unrelated and that there is no attempt to integrate the databases. Thus, the user is aware of the existence of a number of separate databases and the distributed database system would offer the user a selection of schemata - one for each database. Queries would not be distributed, but rather would be directed to the relevant database system as illustrated in figure 2.

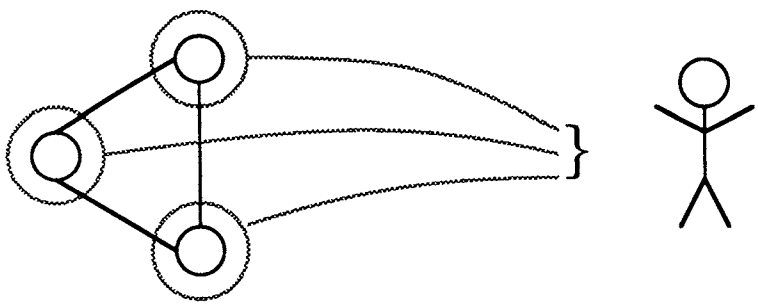


Figure 2. Uniform Access to a Collection of Database Systems

It might be the case that the database systems reside at different nodes of a network of computer systems and, therefore, are physically distributed; or, it might be that the different database systems reside on a single computer system and, therefore, that the distribution is logical. This latter situation is often termed a *multi-database system.*

We regard a multi-database system simply as a special case of a distributed database system in which nodes are logical rather than physical entities, and where message passing is internal and the associated communication costs minimal.

Hence, we shall assume that each database management system resides at a separate node, and that there is a reliable communications network capable of transmitting messages between nodes.

Such a system could be regarded as an *information server* in that its purpose is to provide a user with facilities to retrieve information held in a variety of database systems without the need to learn how to use each individual database management system.

By *uniform access*, we mean that the user should be able to view all schemata in terms of the same data model and access all of the databases using the same query language. If all of the database systems are based on the same database management system, then the distributed database system will be *homogeneous* and the user query language for all users will simply be that of the database management system used.

If the database systems are based on different database management systems, then the distributed database system will be *heterogeneous*. The degree of heterogeneity may vary in that it may be that the database management systems are different implementations of the same data model, or, it may be that they are based on different data models. In the rest of this paper, we shall assume the latter.

In a heterogeneous system, one must consider whether the user data model and query language should be the same for all users of the distributed database system, or, whether a user may use the data model and query language of their local database management system with which they are likely to be familiar. Some proposed systems provide full flexibility by supporting both of these: thus, they provide a global data model and query lanaguage which any user may employ, and they support access via a user's local data model and query language.

Thus, the primary requirement of a heterogeneous system would be the provision of some global query language and data model, and support for the relevant mappings between the global query language and data model and the local query languages and data models. In the case where users employ their local query language and data model, then this global query language and data model may be regarded as internal to the distributed database system: the query language translation in such a system is illustrated in figure 3. The adoption of an internal query language and data model avoids the necessity of providing mappings between all possible pairs of query languages and data models that exist in the system.

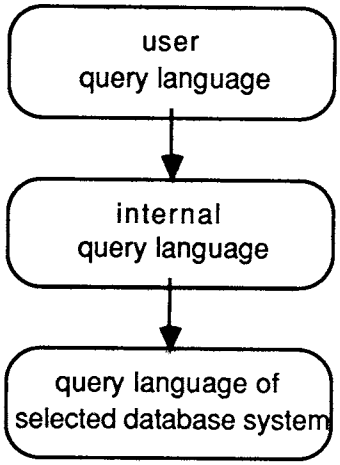


Figure 3. Query Language Translation in A Heterogeneous System

Note that there would also have to be some translation of the format of results so that a result produced by the selected database system could be presented to the user in a form in accordance with their model.

If such a system is to act as an information server, then it is essential that it not only provides support for users to retrieve data from the individual databases, but also that it provides the user with information as to the databases available. It would therefore be desirable to have facilities to allow users to browse both the schemata and the databases.

Since the emphasis in this system is that of permitting users easy access to a selection of databases for the retrieval of information, it is likely that these users would have only read access to data and that all updates would be performed under the direct control of the local database management system.

The second category of system we shall consider is one which provides

access to a single database which is distributed over several nodes.

Thus, we have a single, integrated *distributed database*. The user is presented with a single global schema and queries are directed at the distributed database as indicated in figure 4. The distributed database management system must decompose these queries into a number of subqueries. It then directs these subqueries to the relevant nodes and, on receiving the responses to these subqueries, will compose a response to the initial query. The system supports full location transparency in that a user is unaware of the distribution of data.

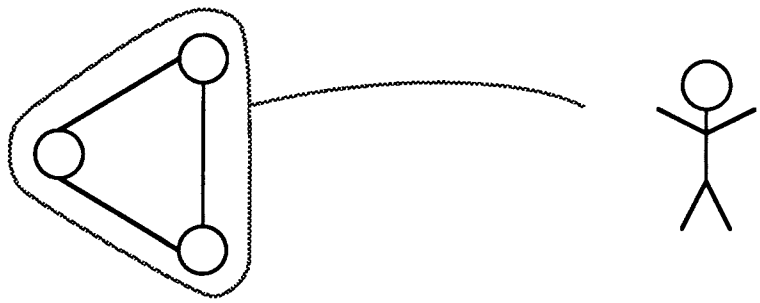


Figure 4. Access to a Distributed Database

The single schema presented to the user is a global view of the distributed database and each node will have a local schema which describes that part of the distributed database held at that node.

Such a system may be developed through the integration of existing database systems which hold semantically related information. The integration may be prompted by the existence of applications which require access to data objects in a number of these databases. It may well be the case that the existing database systems are based on different database management systems then the distributed database system will be heterogeneous. In the latter case, the local schemata must be mapped into a global data model before integration.

On the other hand, it is possible that the distributed database has been designed as such from scratch. For example, a company may wish to have a database system and, for reasons of geographical organisation, performance etc., have decided that the database should be distributed. In this case, it is likely that the system will be homogeneous in that the database management systems at each node will be the same. The design of the distributed database will start with a global schema and then decide on data distribution.

For reasons of performance and availability of data in the event of node or communication failure, a copy of a data object may be held at more than one node of the distributed database. This causes problems in ensuring consistency among the copies of a data object in the event of updates to the distributed database. In addition, the processing of queries will involve selecting which copies of data objects should be accessed to optimize performance.

Hence, the issues of data distribution, query processing and the general management of transactions are much more complicated in the case of a distributed database.

Our third category of distributed database system is in fact a generalization of the first two, and may be described as a system which provides

uniform access to a collection of distributed databases.

By restricting our collection to one, then we simply have the case of a single distributed database as in our second category. If we regard a non-distributed database as a restriction of a distributed database, then our first category would also be a restriction of this category.

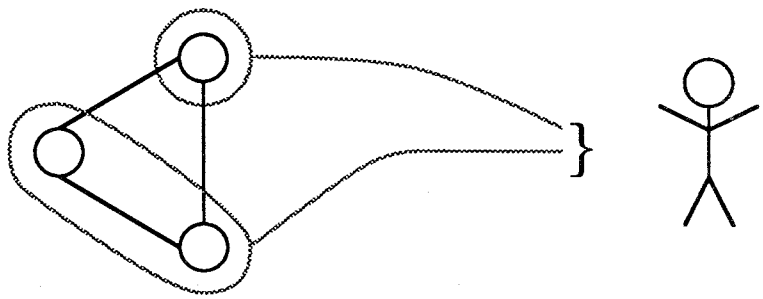


Figure 5. Access to a Collection of Distributed Databases.

If we consider extending the notion of an information server, then one could envisage a system which would provide an environment to support access to a variety of databases available on a computer network. Some of these databases would contain semantically related information and there would be some form of loose integration which would allow users to retrieve information across databases. The integration would be loose in that each node would still retain full control over their database. Thus, one could have a fairly flexible system to which new users and new databases could be added at any time.

Having considered some of the forms of system that might be classified as distributed database systems, one can see that the requirements are dependent on the intended use of the system and its method of development.

The following topics can be identified as important to distributed database systems, although the importance of any one topic to any one system will depend on the characteristics of that particular system:

- data distribution
- query processing
- transaction management
- global data models and laguages
- schema integration.

In the following sections, each of these topics will be discussed in turn.

Whatever the requirements of any one particular distributed database system, there are some general comments that can be made on the architecture of a distributed database system.

A *distributed database management system* (DDBMS) is the software level "on top of" the DBMSs that supports the functions of the distributed database system.

Control of the distributed database system may be centralized, in which case, all nodes communicate via a designated control node. Ultimately, this control node will be responsible for transaction management.

Alternatively, control may be decentralized, in which case, each node has a copy of the DDBMS as illustrated in figure 6. It may be that each node has a transaction manager responsible for transactions initiated at that node, and a data manager responsible for access to the local database.

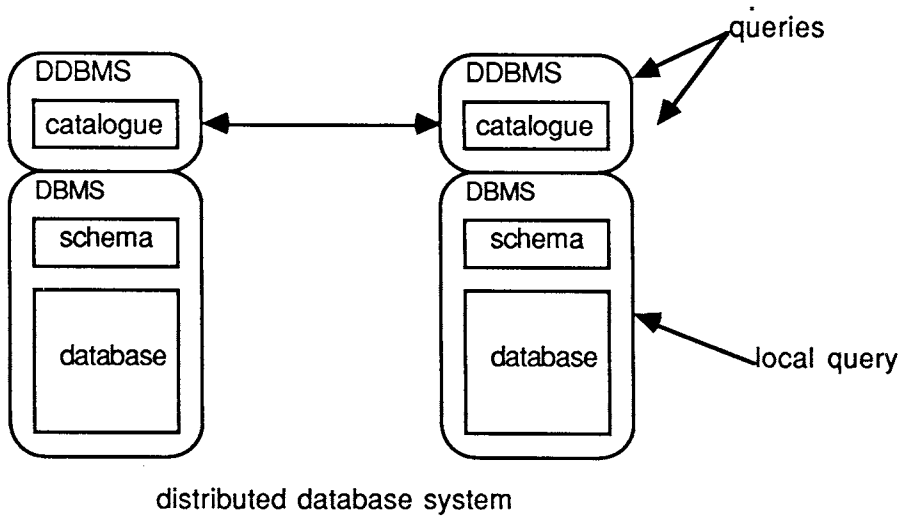


Figure 6. General Architecture of A Distributed Database System

A DDBMS requires information on the local database systems and the distribution of data and this is held in a catalogue.

A distributed databae system may be such that it supports both local and global processing as indicated in figure 6. Thus, there may be local applications which access the local database system directly, and are "unaware" of the existence of a distributed database system.

3. Data Distribution

In a distributed database, the data corresponding to a single global schema will be distributed across a number of different database management systems. In this section, we shall discuss the forms that this distribution of data may take, and the criteria used in the design of a distributed database to decide on the distribution of data.

Note that in the case of a distributed database which is the result of the integration of existing databases, one may regard the distribution of data as predetermined. However, it still may be desirable, either during or after integration, to alter the distribution of data for reasons of performance.

Consider a database as a collection of data objects, and that each data object is associated with a type. For example, given a relational database system with a relation scheme

PERSON (NAME, SALARY, POST)

then we could consider that each tuple of a relation of this scheme is a data object of type PERSON. Then data distribution is concerned with how the data objects are distributed; in other words, which data objects are held at each node.

One of the main objectives in deciding on the distribution of data is that of maximizing processing locality. By this we mean that, since the communication overheads of access to a data object at a remote node are high, the number of such remote accesses should be minimized and the number of local accesses maximized. Other factors that may be taken into account are those of availability of data objects in the event of node and communication network failures, workload distribution and the storage facilities at each node.

To determine the optimal data distribution, the characteristics of applications must be analysed. For each application, one must estimate the frequency of activation of that application at each node, and the number, type and statistical distribution of accesses to data objects. If a large number of the expected applications will be ad hoc interactive user queries, then, clearly, it may be more difficult to predict the application characteristics.

The distribution of data may involve replication, partitioning and fragmentation as discussed below.

With replication, copies of the same data object may be located at more than one node. This may be done for reasons of performance in that it can be used to increase the number of cases in which an application has a local copy of a required data object, and also for reasons of availability in that it can ensure that a copy of a data object is accessible even in the event of a node failure.

Partitioning involves the partitioning of the global schema into a number of local schemata - one for each node. It corresponds to selecting which types of data objects will be held at each node. Thus, we might decide that all data objects of type PERSON should be held at one node while all data objects of type DEPARTMENT would be held at another node.

Fragmentation is similar to partitioning in that it is a matter of deciding which data objects should be stored at which nodes. However, it differs in that one is forming fragments of one particular type of data object and allocating fragments to nodes. The formation of fragments is based on a logical grouping of data objects within a particular type. Thus, we might decide to form two fragments of PERSON objects - one comprising all PERSON objects for persons in administrative posts and one for all persons in research posts.

The process of fragmentation may be split into two stages: the first is the formation of fragments and the second that of the allocation of fragments to nodes - and this allocation may involve the replication of fragme, while the second is concerned with the physical placement of data.

Most of the work on data distribution has been concerned with the fragmentation (and replication) of relations. In part, this is due to the general popularity of the relational model, but it is also due to the fact that relations are more amenable to fragmentation techniques, than, say is a network structure. A relational database is already "partitioned" into discrete structures - namely, individual relations - and there is only one kind of structure to "fragment". Therefore, our discussion of fragmentation will continue in the context of a relational system.

PERSON

NAME	SALARY	POST
Jones F	50000	administrator
Martin M	12000	researcher
Smith W	20000	administrator
Thomas M	18000	researcher
Watson J	25000	administrator

Horizontal fragments of a relation are formed by taking subsets of the tuples based on some selection conditions. These fragments should be complete in that every tuple should belong to some fragment, and be disjoint in that no tuple should belong to more than one fragment. So in our example, we could form two horizontal fragments, P1 and P2, based on the selection conditions

POST = "researcher"

and

POST = "administrator"

giving the fragments:

Fragment **P1**

NAME	SALARY	POST
Martin M	12000	researcher
Thomas M	18000	researcher

Fragment **P2**

NAME	SALARY	POST
Jones F	50000	administrator
Smith W	20000	administrator
Watson J	25000	administrator

One must be able to reconstruct the global relation PERSON from the fragments: in the case of horizontal fragments, this is done by taking the union of all the fragment relations.

Horizontal fragmentation is based on a set of selection predicates. These predicates are determined by examining each application in turn, and, introducing new predicates that will be significant to that application. For example, the formation of fragment P1 based on the predicate POST="researcher", should indicate that there is some application which will access only those tuples associated with researchers.

Having formed the fragments of a global relation, one then considers allocation of fragments to nodes. In the case of non-replication, a fragment would be located at the node with the maximum number of expected references to that fragment. Holding copies of a fragment at additonal nodes will reduce the cost of retrieval accesses; however, since an update must be performed on all copies of a fragment, it will increase the cost of update accesses. Hence, with replication, a fragment should be located at nodes such that the estimated benefit in terms of read accesses outweighs the estimated increase in the cost of update accesses.

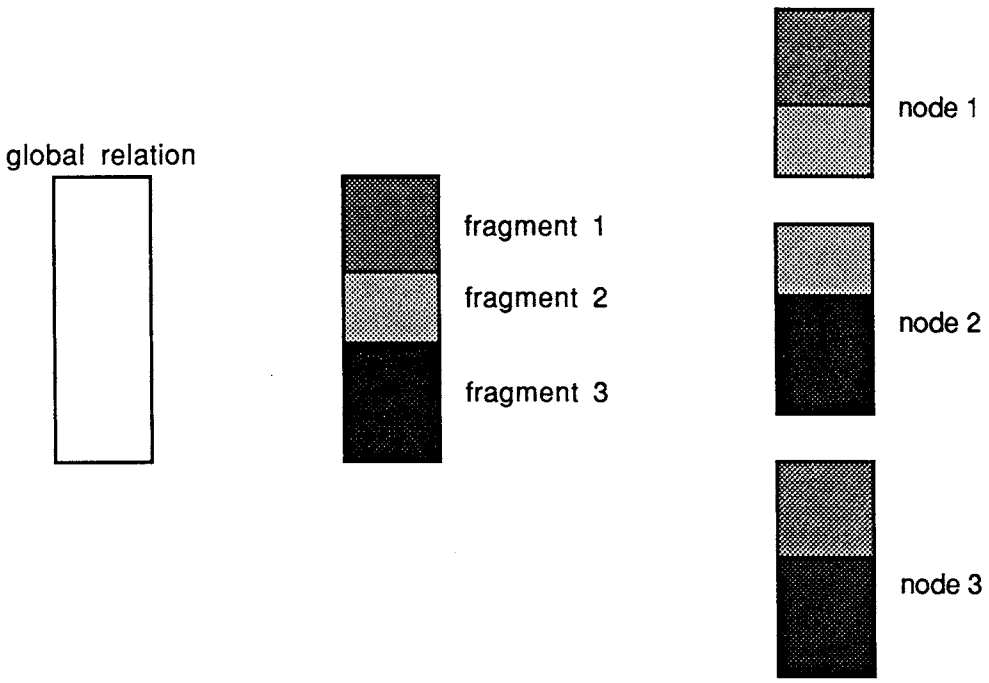


Figure 7. Horizontal Fragmentation and Allocation of a Global Relation

Figure 7 shows a global relation which is split into three horizontal fragments: these fragments are allocated to the three nodes with replication. The fragments of a global relation located at a particular node are said to form the *physical image* of the global relation at that node. For example, in figure 7, the physical image of the global relation at node 1 is the union of fragments 1 and 2.

One can also form vertical fragments of a relation by taking subsets of the relation attributes. The join operation can be used to reconstruct the global relation from the vertical fragments; however, to facilitate this, it is necessary that either the

relation key is included in each fragment, or, that a special tuple identifier attribute is appended to each tuple. For example, in our PERSON relation we could form the two vertical fragments:

NAME	SALARY
Jones F	50000
Martin M	12000
Smith W	20000
Thomas M	18000
Watson J	25000

NAME	POST
Jones F	administrator
Martin M	researcher
Smith W	administrator
Thomas M	researcher
Watson J	administrator

assuming that NAME is a key attribute. Alternatively, if the system used tuple identifiers, then we could have two vertical fragments as follows:

TUPLE-ID	NAME	SALARY
1	Jones F	50000
2	Martin M	12000
3	Smith W	20000
4	Thomas M	18000
5	Watson J	25000

TUPLE-ID	POST
1	administrator
2	researcher
3	administrator
4	researcher
5	administrator

In both these cases, there is no loss of information in forming the fragments, and the global relation can be formed from the natural join of the two fragments. Hence, vertical fragmentation must be complete in that each attribute must be in some fragment, but they need not be disjoint in that the key attribute might appear in all fragments.

Some systems allow overlapping of attributes in vertical fragments other than specifically key attributes. In our example, it would seem that even if NAME were not a key attribute, that it would be desirable to have it appear in both vertical fragments since it seems likely that most applications would require access to the NAME attribute along with the other information held in the fragment. Allowing this replication of data within fragments, creates problems of ensuring consistency in the event of updates, and, for this reason, it is advisable that it only occurs with attributes for which updates are rare.

It is possible to combine horizontal and vertical fragmentation. Usually, systems which do support this, only do so in a limited form by, for example, restricting the fragmentation process to vertical fragmentation followed by horizontal fragmentation - rather than allowing any combination to any level.

It is rare for a distributed database management system to support the full range of distribution options that have been described. Some support fragmentation but not replication. Fragmentation may be limited to either horizontal or vertical - but not both.

As we shall see in the next section, designing a good scheme for data distribution based on predicted application characteristics can be of great benefit in facilitating reductions in the costs of query evaluation.

4. Query Processing

There are two main stages involved in processing a query on a distributed database: the first stage is concerned with the production of a query expression in terms of local database schemata and the second stage is concerned with the selection of evaluation strategies for that expression. Thus, if the query expression established in the first stage, specified a join between two relations R1 and R2, where R1 and R2 are situated at different nodes, then, in the second stage of processing, a strategy for evaluating that join will be selected. For example, it would be possible to move relation R1 to the node of R2 and perform the join there, or vice versa, or, as we shall discuss later, select some other strategy.

In both stages, techniques may be used to optimize performance based on knowledge of data distribution, and, estimated procesing and communication costs. However, it should be remembered that complex optimization techniques are themselves time-consuming. Therefore, in the case of queries that will be submitted many times, it may be worth employing sophisticated query compilation techniques; but with a one-off interactive query, then the query may be evaluated employing only relatively simple and fast optimization procedures.

It is also worth noting what is used as a measure of performance in query optimization. If the nodes of the system are connected by a wide-area network, then the times for message transmission between nodes will be considerable and therefore the communication costs will be the dominant consideration in query optimization. In fact, in such cases, the local processing times are often ignored. In the case of a local area network, where the times for message transmission are far less, then the local processing times are much more significant and should be taken into account. Also, one should consider whether the performance measure should be the estimated total cost of all communication, or, the estimated elapsed time between query initiation and response arrival.

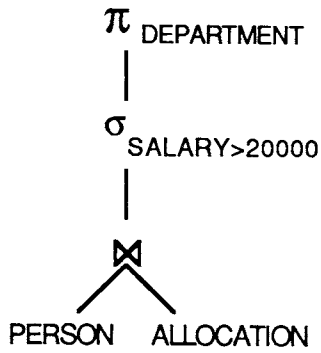
Again, most of the work on query optimization techniques applies to relational systems. To describe the stages of query processing and present an introduction to optimization techniques, we shall use a simple example of a relational system with the two relation schemes:

PERSON (NAME, SALARY, POST)
ALLOCATION (NAME, DEPARTMENT).

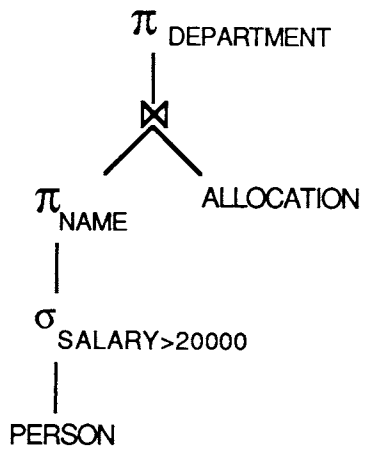
Assume two horizontal fragments of PERSON, and three horizontal fragments of ALLOCATION. Now, suppose further that we have a query to list the names of departments containing a person whose salary is greater than 20000. We shall express this in relational algebra as follows:

$$\pi_{DEPARTMENT}\,(\sigma_{SALARY>20000}\,(PERSON \bowtie ALLOCATION)\,).$$

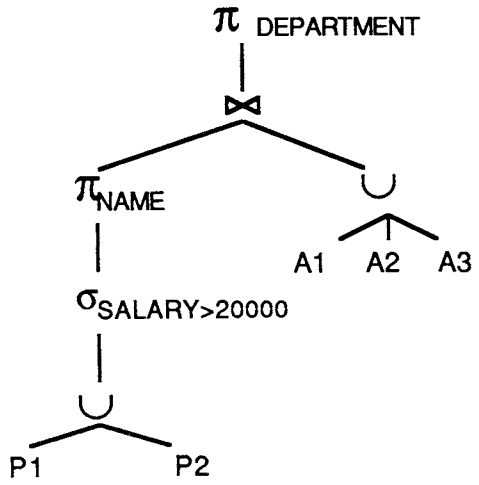
This expression can be represented by the following query evaluation tree:



With all relational algebra expressions, whether for distributed or non-distributed databases, a general rule is to reduce the operand relations as much as possible before performing a join operation. In simple terms, this is achieved by moving projections and selections towards the leaves of the query tree, and introducing additional projections, whenever possible, to remove attributes no longer required in the query execution. In this way, our query tree would become:

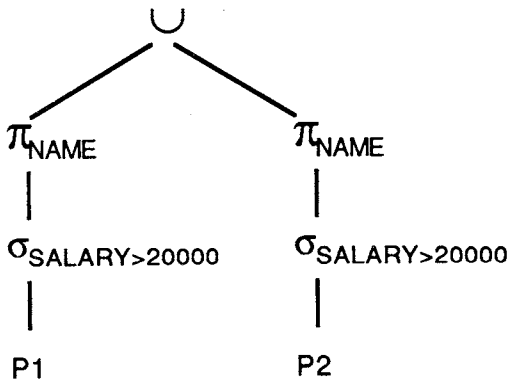


Now, as it stands, our query is expressed in terms of the global relations PERSON and ALLOCATION and we must transform it to an expression in terms of the relation fragments i.e. for global relations which are horizontally fragmented, we replace a reference to the relation by an expression of the union of the fragments. Our query tree would then become:



We now have a query expression that could be evaluated by fetching all the fragments, constructing the global relation and evaluating the query. However, this would be considered the worst case in terms of performance because of the large associated communication costs. So we look for further equivalence transformations which would improve performance by reducing the communication costs.

Unary operations, such as project and select, can be processed locally i.e. at the node of the operand relation. However, operations, such as join and union, which involve more than one operand relation, will require tuples to be transmitted between nodes when the participating relations are at different nodes. To minimize the quantity of data to be transmitted, it is best to distribute as much processing as possible to be performed locally. Moving selections and projections "below" unions and joins in the query tree is therefore advantageous. For example, if we consider the processing performed on the fragments of PERSON, the projection and selection may be distributed over the union of P1 and P2 as follows:



and is beneficial by reducing the size of the operand relations.

The next stage in the query optimization process is to determine whether any of the branches of the query tree can be removed i.e. whether any of the processing can be

eliminated. Such subtree removal may take place if we can determine that the relation at the root of the subtree would always be empty.

This can be done by associating a qualification with each relation : this qualification is a condition that will be satisfied by all tuples of that relation. By starting from the leaves of the query tree and working towards the root, the algebra of qualifications can be applied for each operation and thus we can determine the qualification associated with each node relation in the tree. In the case where a qualification condition is contradictory, then the corresponding relation is intrinsically empty, and the subtree with that relation as root can be removed since it will not contribute to the final result.

In our example, we could express the qualification for the leaf P1 as

$$[\ P1 : POST = \text{"researcher"}\ ]$$

then applying the selection

$$\sigma_{SALARY>20000}$$

we can refine the qualification as

$$[\ \sigma_{SALARY>20000}(P1) : POST=\text{"researcher"} \ \&\ SALARY>20000\ ].$$

If we continued evaluating the qualifications for our example, it would not lead to any pruning of the query tree. However, if, for example, relation PERSON had been fragmented, not only on the value of the POST attribute, but also, on the value of the SALARY attribute, then we might have had a fragment P1' with qualification

$$[P1' : POST=\text{"researcher"} \ \&\ SALARY<15000\ ].$$

Then, by applying the selection, the qualification would become

$$[\sigma_{SALARY>20000}(P1') : POST=\text{"researcher"} \ \&\ SALARY<15000 \ \&\ SALARY>20000].$$

We would have a contradictory qualification and the relation at the root of the subtree intrinsically would be empty and we could prune the query tree accordingly.

Note that since horizontal fragmentation is based on the analysis of applications, then, in practice, such simplifications should occur frequently. In other words, many applications would require access toonly a limited number of fragments.

Just as projections and selections can be moved to nodes by moving them "below" union operations in the query tree, it is possible to distribute join operations in the same way.

$$(A1 \cup A2) \bowtie (B1 \cup B2) = (A1 \bowtie B1) \cup (A1 \bowtie B2) \cup (A2 \bowtie B1) \cup (A2 \bowtie B2)$$

At first glance, this may not appear to be of much benefit - but again we would hope that some of the results of these "sub-joins" would be empty intrinsically and therefore could be eliminated. Again, such decisions can be based on the use of

relations with qualification. If both PERSON and ALLOCATION were fragmented horizontally on the basis of names, then, clearly ,the join of a PERSON fragment with names in the range "A" to "H", and an ALLOCATION fragment with names in the range"N" to "S", will always yield the empty relation.

This elimination of sub-joins is aided by the use of derived horizontal fragmentation where a relation is horizontally fragmented based on the fragmentation of another relation. For example, if our applications were such that access to relation ALLOCATION is usually in conjunction with access to relation PERSON, then we could have a horizontal fragmentation of ALLOCATION which corresponded to the fragmentation of PERSON rather than one which came solely from a selection condition on the values of attributes of ALLOCATION, as follows:

P1

NAME	SALARY	POST
Martin M	12000	researcher
Thomas M	18000	researcher

A1

NAME	DEPARTMENT
Martin M	computing
Thomas M	economics

P2

NAME	SALARY	POST
Jones F	50000	administrator
Smith W	20000	administrator
Watson J	25000	administrator

A2

NAME	DEPARTMENT
Jones F	computing
Smith W	personnel
Watson J	economics

Having examined the techniques for the transformation of query expressions, we now assume that we have a query tree which is ready for evaluation. The problem is one of selecting a strategy for the evaluation that will minimize the communication costs.

Consider a join operation between PERSON and ALLOCATION where the two relations reside at different nodes as illustrated in figure 8:

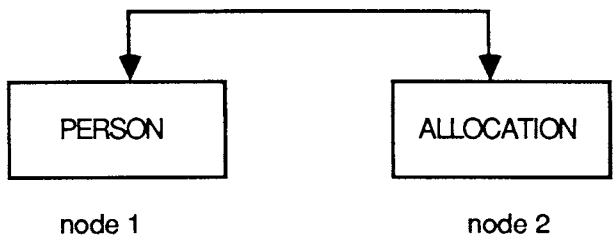


Figure 8. Placement of Relations to be Joined

One strategy would be that of moving relation ALLOCATION to node 1 and then performing the join at node 1; while another strategy would be to move relation PERSON to node 2 and then perform the join at node 2. The choice between these two strategies could be made by considering the size of the relations and also the required destination of the result. However, it still may be the case that all tuples of

a relation are transmitted to the other node and many of them will not participate in the join i.e. there are no tuples in the other relation with matching values for the join attributes. A technique that would select only those tuples that would actually participate in the join is required.

Consider our example of performing a natural join of PERSON and ALLOCATION and assume that the join will be performed at node 2. Then those tuples of PERSON that will participate in the join are those with a value of attribute NAME that appears in some tuple of the ALLOCATION relation. By projecting ALLOCATION to NAME, we would obtain a relation that contained all the NAME values that occurred in relation ALLOCATION. This projected relation could then be transferred to node 1, and by joining it with relation PERSON, we effectively would select out those tuples of PERSON which have NAME values that appear in ALLOCATION. Now we need transfer only those selected tuples to node 2 and perform the join with ALLOCATION.

The set of selected tuples of PERSON that would participate in the join with ALLOCATION is known as the semi-join of PERSON and ALLOCATION and is denoted

$$\text{PERSON} \ltimes \text{ALLOCATION}$$

It may appear that we have gained nothing and in fact may have increased evaluation costs by introducing additional operations. However, in the case where relations are large, both in terms of number of attributes and number of tuples, then great savings can be made in the quantity of data transmitted in forming the join. Having stated that, further consideration of the above example would indicate that there are cases where savings are unlikely to be great (if anything) because the nature of the relations involved are such that most of the tuples will participate in the join. An additional constraint that every person must be allocated to at least one department would mean that all tuples of PERSON and ALLOCATION would participate in a join of the two relations. Such information is useful in determining whether the construction of a semi-join would be beneficial in reducing communication costs.

The effective use of optimization techniques in both the transformation of query expressions, and in the selection of evaluation strategies is dependent on the availability of knowledge about the properties of relations. The more detailed the knowledge that is available, the better the estimates of communication and processing costs will be, and hence there wll be an improved basis for selection.

We have already discussed the idea of associating qualifications with relations in query transformation. Another general technique is that of storing a profile of each relation. Basically, these profiles describe the size of the relation in terms of the number of tuples, the number of attributes and the sizes of the attribute values, and information on the number of different values of an attribute that appear in the relation. This information can then be used to estimate the size of relations to be evaluated and this can then be used to estimate the costs of transferring a relation between nodes.

Our discussion of query evaluation has assumed that each node is capable of receiving a relation and performing the specified operations. A problem that may arise in heterogeneous systems is that of different nodes having different processing capabilities. For example, a particular relational database management

system may not be capable of accepting input relations for processing, or, it may not be capable of performing all the required operations. Hence, in deciding upon a strategy for evaluation, the processing capability of each node should be taken into account.

5. Transaction Management

One of the fundamental roles of any database management system is that of ensuring that different applications can access the same database without interference from each other. If an error condition occurs during the processing of an application, then the effects of that error should not be propagated to other applications being processed at the same or a later time. To this end, the system has atomic units of processing which are termed transactions.

A transaction must either complete successfully and have its effects permanently recorded in the database by means of a *commit* operation, or, if it cannot complete successfully due to an error or system failure, then its effects must be completely removed by means of an *abort* operation. To be able to do this, the effects of a transaction should be visible to other transactions only after that transaction has committed. In effect, this means that transactions run in isolation.

Once a transaction has committed, then the system must ensure that the effects will not be lost in the event of a system failure or disc crash. It therefore requires mechanisms to recover the database to a current and consistent state in the event of such failures. The system may maintain some form of log record of transactions and this can be used to ensure that the effects of any transactions that have committed are *redone* and the effects of any transactions that had not committed at the time of failure are *undone*.

If transactions are to execute in isolation, then the outcome of executing a set of transactions concurrently should be the same as if those transactions had been executed in some serial order. Thus, two concurrent transactions, which access some common data objects, must somehow synchronize their activities i.e. there must be some form of concurrency control.

Basically, a concurrency control mechanism is a method of ordering transactions and ensuring that the effect of the transactions is the same as if those transactions had executed serially in this order. The ordering could be based on one of the following: the order of intitiation of transactions, the order in which transactions access data objects, or, the order of completion of transactions.

To provide a serialization ordering based on order of initiation, transactions are allocated a unique sequence number called a *timestamp* at intiation. If two transactions have timestamps T_1 and T_2 such that $T_1<T_2$, then the transaction with timestamp T_2 will be referred to as the "younger" transaction in that it occurs after the transaction with timestamp T_1 in the ordering.

When a transaction requests access to a data object, the system checks whether this access would violate the ordering scheme. In the case of a requested read access, the

system must check that no "younger" transaction has already written to that data object, while with a requested write access, a check must be made that no "younger" transaction has already written or read that data object. If a check fails, then the requesting transaction must abort and restart. In fact, in the case of a transaction requesting a write access to a data object which has already been written to by a "younger" transaction, and the data object has not been read by any "younger" transactions, one may regard this as a transaction trying to place obsolete information in the database and, instead of aborting the transaction, simply ignore the request.

So, for each data object, the system must record the timestamps of the last transactions to have read and written that data object.

In a distributed database system, transactions may be initiated at different nodes. We will assume that each node has an associated transaction manager which will maintain its own transaction sequence number counter. To ensure that the transaction timestamps are unique throughout the system, the transaction timestamps will be generated from the node sequence number and the node identifier as illustrated in figure 9.

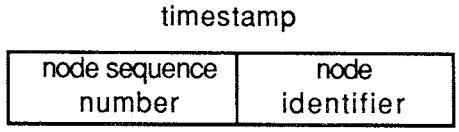


Figure 9. Timestamp Generation in A Distributed Database System

To maintain a certain degree of "fairness" among transactions from different nodes, it is desirable to try and keep the sequence number counters for the nodes loosely synchronized: otherwise, a node which generated lots of transactions would have a high sequence number count and its transactions would be at an advantage over transactions from a node with a low sequence number count. A simple remedy is to have nodes update their sequence number counts on receipt of a timestamped message from a node with a higher sequence number count.

The severity of aborting and restarting transactions in such a timestamping scheme has led to the proposal of a scheme which instead of checking whether accesses will violate the serialization order, actually enforces this order by deferring access to objects until it is known that all accesses from "older" transactions have been dealt with: this adaptation is known as *conservative timestamping*. The problem is now one of knowing that there will be no future requests from "older" transactions - and ensuring that transactions are not made to wait forever.

Assume that each transaction manager sends access requests for its transactions in timestamp order, and that the communications network delivers messages in the same order in which they are sent. Then if a transaction manager has received from each node an access request with a timestamp greater than T, then the transaction manager knows that there can be no future request with timestamp <T and, therefore, that it can grant access to a transaction with timestamp T. A transaction

manager with no recent access requests to a particular node may send a timestamped null request message indicating the timestamp of its last access request: this ensures that no transaction should wait unnecessarily. To some extent, this method shifts most of the problem to the transaction managers in that they must have some way of guaranteeing to send access requests in timestamp order.

Another method of ordering transactions is that of using the order in which they access data objects to determine the serialization ordering. If two transactions access the same data object X, then whichever transaction accesses X first will precede the other transaction in the ordering scheme. Thus, the second transaction must be prevented from accessing that object until after the first transaction has completed. Access to data objects would be controlled by some form of locking mechanism. The first transaction will hold a lock on X until it commits.

This ordering of transactions is partial rather than total in that two transactions which access no common data objects will not necessarily be ordered with respect to each other.

A problem that may arise with this dynamic form of ordering is that of *deadlock*: this corresponds to a situation in which two conflicting orderings have been determined. Consider two transactions T_1 and T_2 which both access data objects X and Y. It is possible that the concurrent execution of these two transactions could lead to the situation, illustrated in figure 10, in which both transactions would wait forever.

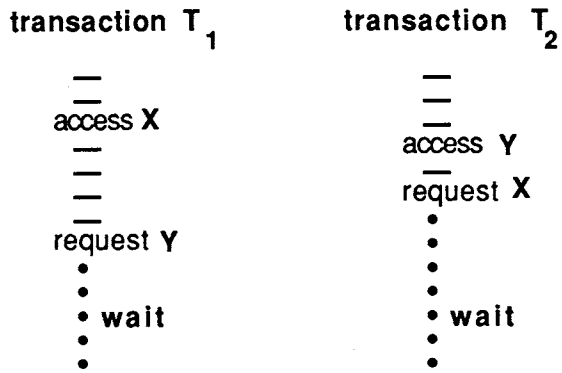


Figure 10. Transaction Deadlock

The problem is that T_1 accesses X before T_2 and that implies an ordering $T_1<T_2$; however, T_2 accesses Y before T_1 and that implies an ordering $T_2<T_1$. Clearly, these two orderings conflict and the result is a deadlock situation.

Deadlock can be detected by maintaining a *waits-for graph* which represents transactions' locks on data objects and requests for data objects. Figure 11 shows the waits-for graph for our transactions T_1 and T_2 in their deadlock situation.

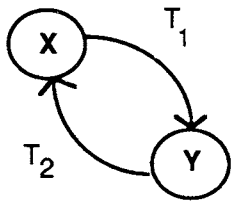


Figure 11. A Waits-For Deadlock Detection Graph

A cycle in a waits-for graph indicates that the set of transactions involved in the cycle are "waiting on each other" and none can proceed. If deadlock is detected, then one or more of the transactions must be aborted and this will free some of the locks on data objects.

Deadlock in a distributed system may involve more than one node and it is therefore necessary that transaction managers exchange information on their local waits-for graphs. One possibility would be to have centralized deadlock detection, in which one node was responsible for maintaining a global waits-for graph and detecting and recovering from any global deadlock situations. Each transaction manager would send that part of its waits-for graph relevant to global deadlock to the node responsible for global deadlock detection. The part of a local waits-for graph relevant to global deadlock is that involving potential cycles with transactions waiting for external data objects, or external requests for data obejcts at that node. For example, the local waits-for graph in figure 12 could form part of a global deadlock situation and would therefore be sent to the node responsible for global deadlock detection. All such local waits-for graphs received by that node would be combined to form the global waits-for graph.

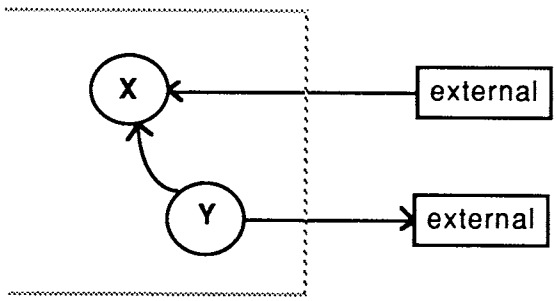


Figure 12. Part of a Local Waits-For Graph Indicating Possible Global Deadlock

Other methods for global deadlock detection have been proposed that will avoid centralization. It may be that any transaction manager can detect global deadlock through a continual exchange of relevant information among the nodes: a form of distributed deadlock detection rather than centralized daedlock detection results.

In a system which supports replication of data objects, the basic locking scheme for concurrency control will have to be adapted to ensure that there is consistency among all copies of a data object. A simple scheme would be to require a transaction to hold locks on all copies of a data object for a write access and to hold a lock on a

single copy for a read access. This would prevent one transaction writing to a data object while another transaction was reading another copy of the same data object. One variation is that of majority locking in which, for any access, a transaction must obtain a lock on a majority of the copies of the data object. Another method is that of designating a primary copy for each data object and before accessing a copy of that data object, a lock must be obtained on the primary copy.

Ordering transactions according to their order of completion may at first seem strange. The idea is that all transaction processing is performed on local copies of data objects and that the effects of a transaction are recorded in the database only during the commit operation. Therefore, before performing the commit operation, the system checks to see whether the transaction is in conflict with any transactions that have already committed. For each transaction, the system maintains a record of all objects read and written by that transaction. At the end of transaction processing, the read and write sets of the transaction are compared with those of transactions that have committed while this transaction was executing. If there is a non-empty intersection of, say, the read set of this transaction and the write set of a committed transaction, then a conflict exists and this transaction must be aborted and restarted. If there is no conflict detected during this validation procedure, the transaction is allowed to commit and its effects are recorded in the database by performing the appropriate write operations.

In some distributed database systems, the transaction manager for a transaction may invoke sub-transactions which will execute at other nodes. Then the transaction manager must ensure that either all of the sub-transactions commit during the commit phase, or, that none of them commit. This is achieved by some form of two-phase commit protocol. During the first phase, the transaction manager establishes whether or not each subtransaction is ready to commit. If all subtransactions are ready to commit, then the transaction manager will issue a commit command to all subtransactions. However, if any one of the subtransactions fails to indicate that it is ready to commit, the transaction manager will issue an abort command to all subtransactions.

The selection of concurrency control and recovery mechanisms for a particular distributed database management system depends on both the general requirements of the system and its intended applications. In general, concurrency control mechanisms based on locking are appropriate in systems where the probability of conflict between transactions is high: this is mainly due to the high overheads associated with transaction aborts and restarts that exist with other mechanisms. However, it must be remembered that there may be high overheads associated with lock maintenance and that locking may be very restrictive.

Any mechanism that has some degree of centralized control would be considered undesirable in a system where local autonomy is of great importance, and it also has disadvantages in terms of possible communication bottlenecks and vulnerability to node failure. However, in a star-structured system in which all communication is via a central node, then such mechanisms may be a natural choice.

Many techniques for concurrency control and recovery have been proposed and it is certainly one of the most active areas in database research. Here, we have been able to give only a very brief overview of some of the approaches. For details of

concurrency control and recovery in both distributed and non-distributed database systems see [2].

6. Global Data Model and Query Language

In a heterogeneous distributed database system, the choice of a global data model and query language is one of the major design issues. As stated previously, the suitability of any particular model and language will depend on whether they are intended purely as internal to the system or as a user model and language. In the latter case, clearly the model and language must be user-oriented, while, in the former case, the emphasis would be on issues such as ease of representation and mappings, and query processing capabilities.

In any case, the model and language must be capable of representing a wide range of data models and languages covering both record and object oriented data models. The global model must be able to capture both the structure and semantics of the different schemata, and the global language must be able to represent the operations of the various query languages. It should be accepted that there may be parts of a query which cannot be translated - and that these will be treated as exceptions.

Some form of extended relational model seems to be a popular choice - although it should be stressed that this is an area of research which is still at a fairly early stage due to the fact that there has been less research effort placed on heterogeneous distributed database systems in the past. An extended relational model is one in which relations are used to represent both data and metadata i.e. relations are used to represent both the schema and the data. Thus, special metadata relations are used to describe entity types, relationships, constraints etc.

The popularity of an extended relational model stems from various factors. The lack of structure and inherent constraints makes the model very flexible and suitable for the representation of other models. As can be seen from the earlier sections on data distribution and query processing, relational systems are well-suited to these techniques and most of the work in these areas as been done in the context of a relational system.

In the Proteus project [1], it was considered important that users should be able to use their own local query language and, accordingly, an internal global query language and daat model were used. The internal query language was based on relational algebra, and schema information was represented by a form of extended relational model.

In contrast, the MULTIBASE system [4] provides all users with the same query language and data model, and their choice was based on the functional data model as proposed by Shipman [6].

7. Schema Integration

To form a distributed database from a set of existing databases, one must integrate

the set of local schemata into a single, global schema and produce a set of mappings from this global schema to the local schemata. This integration of schemata is also sometimes used in a method of database design where a number of different user views are combined to form a database schema: in this case, the process is referred to as view integration.

Schema integration is normally associated with heterogeneous distributed database systems - not because of any intrinsic property, but rather because, as already discussed, the heterogeneity often arises out of the integration of existing databases. In the case of a heterogeneous system, clearly the local schemata must be mapped into equivalent schemata of the chosen global data model before integration can take place.

The main problem of schema integration is that of trying to identify semantic equivalences in the local schemata. In other words, one must try to find overlaps in the schemata where the same part of reality is being described.

It is very difficult to identify true semantic equivalences without good knowledge of the application reality. With a tightly-coupled distributed database system with a single administrative body, then it will be the responsibility of that body to take decisions on semantic equivalence and to guide the integration process.

In the case of a loosely-coupled system with no single administrative body, it is much more realistic to accept that there will not be general agreement on a global schema: instead, each node may decide on its own global view of the distributed database, ignoring any information that is not of interest to the users at that node.

To demonstrate some of the difficulties in schema integration, we will consider a very simple example in which we have two local schemata, each with student objects with a set of attributes, as illustrated in figure13.

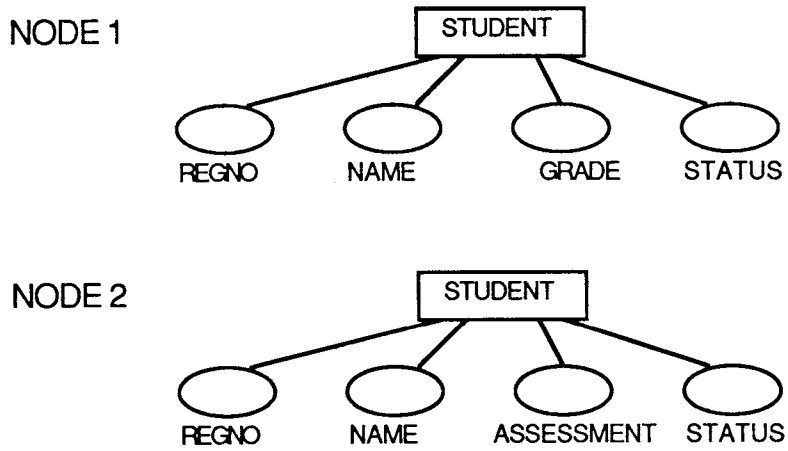


Figure 13. Local Schemata at Nodes 1 and 2

If we accept that student objects are semantically equivalent, then the two local

schemata may be merged into a single tree-structured schema with a generalized object type STUDENT, and the two specialized object types STUDENT-1 and STUDENT-2 corresponding to the two local schemata as given in figure 14.

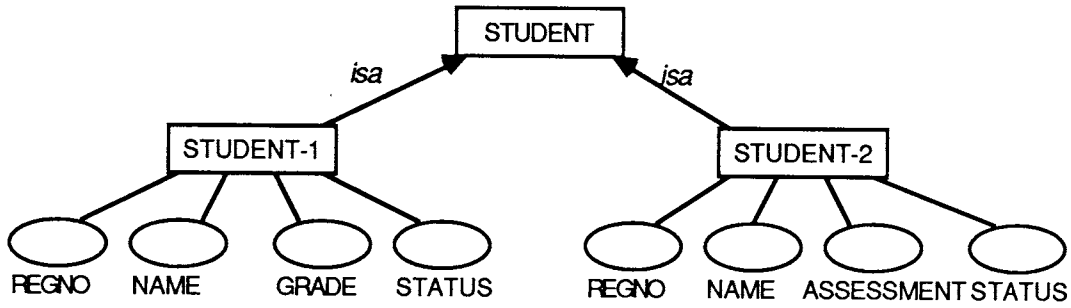


Figure 14. Introduction of a Generalized Object Type

Then the next stage is to examine the attributes of STUDENT-1 and STUDENT-2 and decide which ones are semantically equivalent and can be associated with the generalized type STUDENT.

The first problem to consider is that of naming. It is possible that two attributes may be semantically equivalent but have different names e.g. GRADE and ASSESSMENT. On the other hand, two attributes may have the same name but not be semantically equivalent: for example, in one case, STATUS might be a string identifying the marital status and in the other case be a string identifying student status e.g. postgraduate.

Having decided that two attributes are semantically equivalent, one must consider whether the scope of the attribute is local to that node or global to the distributed database. For example, if we decide that GRADE and ASSESSMENT are semantically equivalent, then consider the interpretation of a value "A" for GRADE at node 1. Does this value have a global interpretation? Would it have the same meaning as a value "A" for ASSESSMENT at node 2?

If attributes of STUDENT-1 and STUDENT-2 are semantically euqivalent and have global scope, then the attributes may be removed from the specialized object types and replaced by an attribute of the generalized object type. For example, we may decide to have a NAME attribute for STUDENT to replace those of STUDENT-1 and STUDENT-2 as given in figure 15.

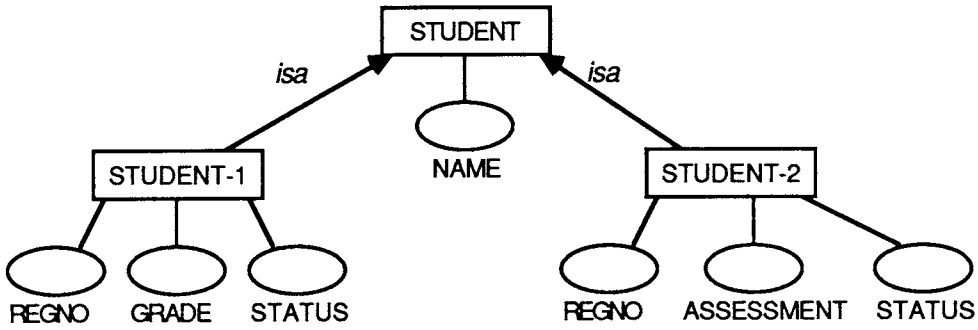


Figure 15. Moving Equivalent Global Attributes to the Generalised Oject Type

If the attributes are semantically equivalent but the scopes are local, then the actions possible will depend upon whether the intersection of the sets of student objects of nodes 1 and 2 is non-empty. If the intersection is empty, then the local attributes may be replaced by a single attribute for STUDENT that refers to the attribute values of student objects in their local environment.

Thus, if there are no students who appear in both the database at node 1 and the database at node 2, then the GRADE and ASSESSMENT attributes of STUDENT-1 and STUDENT-2, respectively, could be replaced by a single GRADE attribute of STUDENT as represented in figure 16. Then for a particular student, the value of attribute GRADE would be interpreted according to the relevant node. Thus, if nodes 1 and 2 were associated with different universities, and it were the case that no student could belong to more than one university, then the value of GRADE would be interpreted in the context of the appropriate university.

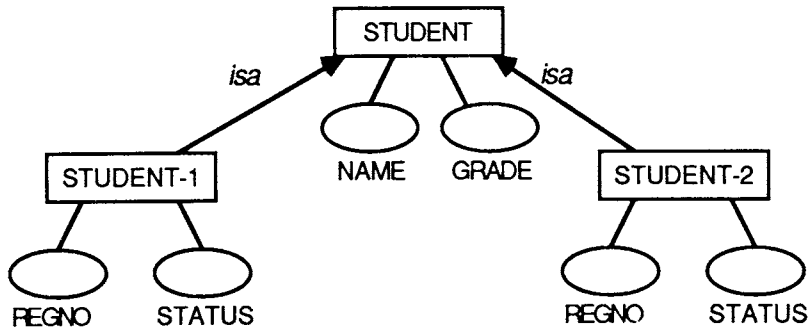


Figure 16. Generalizing Equivalent Local Attributes in Disjoint Databases

However, if it were possible that a student could belong to both universities, then that student would have an associated value for both GRADE and ASSESSMENT. In this case, the attributes must remain with the specialized object types and we could, for example, introduce a new attribute such as AVERAGE-GRADE for the generalized type provided that this could be given some reasonable interpretation.

If the local schemata have key attributes which are used to uniquely identify objects, then in the process of schema integration one must decide upon a global key

attribute. Assume that REGNO is the key attribute in both schemata. If the REGNO attributes have global scope, then the value of REGNO will be unique globally and, therefore, the local attributes REGNO can be replaced by the global attribute REGNO of student.

However, if the scopes are local, then a new global key will have to be devised. One way of achieving this would be to augment the values of REGNO by a local identifier to make them globally unique. In both of these cases, the global schema would be as given in figure 17.

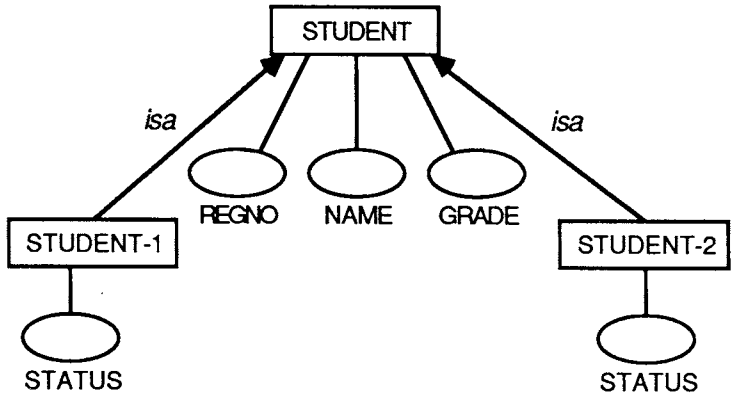


Figure 17. Making REGNO a Global Key Attribute

Alternatively, one could retain REGNO as an attribute for both STUDENT-1 and STUDENT-2 and introduce a new attribute to STUDENT which did have the global key property.

Here, we have illustrated only some of the problems of schema integration. Notice that in our example, we started from two very similar structures for student object types. In practice, two databases may both contain information on students - but the level of detail or structural representation may be very different. In addition, different representations may be used for similar attribute values: for example, currency values may be recorded using different units of currency.

8. Concluding Remarks

An overview of the major issues of concern to the designers of distributed database systems has been presented. We have chosen to discuss general concepts rather than provide descriptions of particular research systems that have been developed.

Two systems that are particularly worthy of mention because of their significance in the historical development of distributed database systems are SDD-1 [5] and Systems R* [8].

SDD-1 was developed by the Computer Corporation of America, and was the first

large distributed database project. The most significant contributions of the project were the techniques developed for concurrency control and query processing.

System R* is a distributed database management system based on IBM's relational database system, System R. One of the major features of the system was that of retaining the local autonomy of nodes.

Both of these systems are homogeneous and relational. Nearly all of the early work on distributed database systems was on homogeneous systems, but recently there has been a great increase in interest in heterogeneous systems. In part ,this is due to the fact that heterogeneous systems were seen as "too difficult" and it was better to concentrate intial research on the simpler homogeneous systems. However, it is also the case that there is now greater appreciation of the potential uses and flexibility offered by heterogeneous systems.

9. References

[1] Atkinson M P et al, *The Proteus Distributed Database System*, in Proc. of the Third British National Conference on Databases (BNCOD 3), ed. J Longstaff, Pub. Cambrdige University Press, 1984.

[2] Bernstein P A, Hadzilacos V and Goodman N, *Concurrency Control and Recovery in Database Systems*, pub. Addison-Wesley, 1987.

[3] Ceri S and Pelagatti G, *Distributed Databases: Principles and Systems*, pub. McGraw-Hill, 1984.

[4] Landers T and Rosenberg R L, *An Overview of MULTIBASE*, Distributed Databases, ed. H J Schneider, pub. North-Holland, 1982.

[5] Rothnie J B et al, *Introduction to a System for Distributed Databases (SDD-1)*, ACM Transactions on Database Systems, Vol. 5 No. 1, 1980.

[6] Shipman D, *The Functional Data Model and the Data Language Daplex*, ACM Transactions on Database Systems, Vol. 6 No. 1, 1981.

[7] Ullman J D, *Principles of Database Systems*, 2nd ed., pub. Pitman, 1983.

[8] Williams R et al, *R*: An Overview of the Architecture*, Proc. of the International Conference on Database Systems, 1982.