Chapter 3

Distributed database systems

Essential reading

"An Introduction to Database Systems", sixth edition, by C. J. Date, published by Addison-Wesley, 1995, [ISBN 0-201-82458-2], *Chapter21* (pp. 593 – 627).

Alternatively

"Database Systems: A Practical Approach to Design, Implementation and Management", second edition, by T. Connolly and C. E. Begg, 1999, [ISBN 0-201-34287-1], *Chapter 19* (pp. 645-685).

Further reading

"Database Systems Concepts", second edition, by H. F. Korth and A. Silbershatz, published by McGraw-Hill, 1991, [ISBN 0-07-100804-7], *Chapter 10, Chapter 11 and Chapter 12* (pp. 313-422) — for a comprehensive and very concise description.

"Database Systems: A Practical Approach to Design, Implementation and Management", second edition, by T. Connolly and C. E. Begg, 1999, [ISBN 0-201-34287-1], *Chapter 20* (pp. 687-726) — for some more specialised topics.

If you want to explore more into this subject, refer to "Principles of Distributed Database Systems", by M. T. Ozsu and P. Valduriez, published by Prentice Hall, 1998 [ISBN 0136597076]

Introduction

Suppose that a company, "Integrated Solutions Ltd.", has different departments located in different cities, say the Personnel and Finance departments in London, the Sales department in Reading and the Development department in Stockley Park (Figure 1). Each department keeps its own data in its own database; i.e., each department has its own database system comprising the data that is most relevant to them.

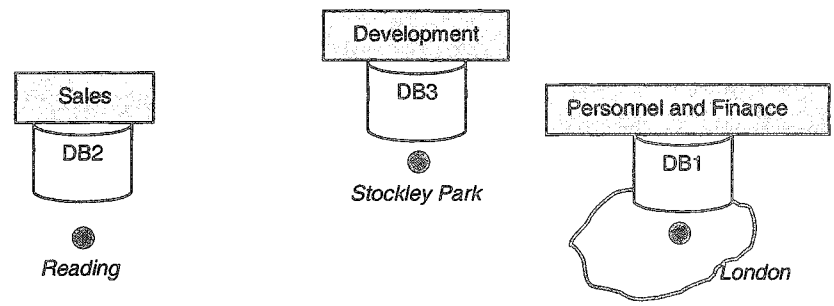

Figure 1: A company having different database systems at different locations

Situations when one department needs access to some data of another department are bound to appear. For instance, the development department might need to consult the sales figures to decide upon its future strategy. The Finance department needs periodical access to the sales figures to update its accounts. The list of examples can continue.

If the three databases are (physically) independent from each other, data stored at one location cannot be directly accessed by the DBMS of another location. In this situation,

the data has to be transferred from the location that stores (owns) it to the location that requires it. This transfer can be made via:

- *printed reports* – the reports are created at the owner site, posted to the site that required this data and manually inputted there;
- *result files* – files of different formats are created at the owner site; the data can be input from there at the destination site.

Other methods could be devised, but they would all require a considerable amount of extra processing resulting in negative consequences: delays, extra costs, higher probability of errors, etc.

However, the other possibility is that the three databases are linked via a communication network. This allows the DBMS at each site to communicate with the other DBMSs, thus having access to the data stored at the other sites. Users do not need to know where the data is stored. They simply interact with the DBMS at their site. It is the DBMS's "responsibility" to identify where the requested data is stored.

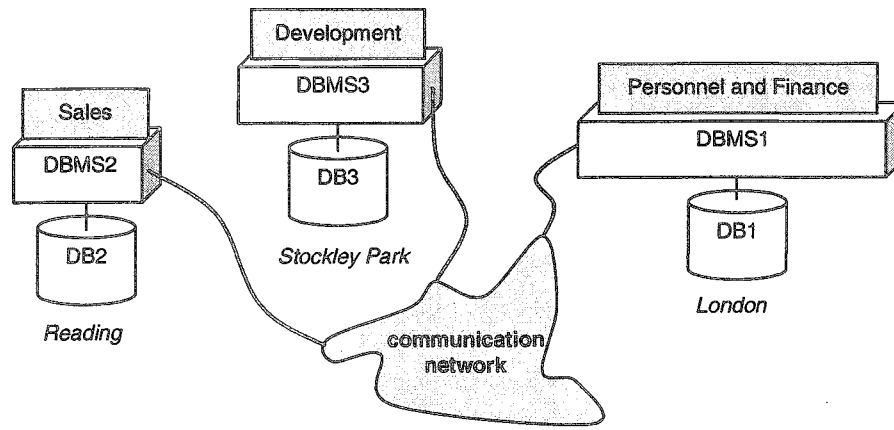

Figure 2: A distributed database system

Definition: A distributed database system is a collection of local database systems such that

- *each local database system can be used independently of the others*
- *all these local database systems are interconnected and communicate with each other, so that*
  - *each local database system can access the other system's data*
  - *from the point of view of the user, this access is made transparently; i.e. the user perceives all data as if it were stored in his/her local database system.*

The location of a "local database" (as used above) is usually called a *site*. From the above definition it can be deduced that

- each site must have a DBMS that manages its own data independently from the others (this is a DBMS as we tacitly understood so far);
- an extra subsystem must exist at each site, that caters for the interaction with the other sites.

It is the combination of these two components that constitutes a Distributed Database Management System (DDBMS).

We have assumed so far that the sites were geographically distributed and interconnected via a Wide Area Network (WAN). This is not always the case. It is possible, for instance, that the different sites exist on (separate) machines at the same location, these machines being connected via a Local Area Network (LAN). Moreover, different sites might even be located on the same machine. Many of the issues raised are

equally relevant whether the sites are distributed geographically (WAN connected) or locally (LAN connected or on the same machine).

In this chapter, we shall assume that the sites are geographically distributed.

Another assumption will be made throughout this chapter, namely that each site runs a copy of the same DDBMS - the *homogeneity assumption*. Without this assumption, all kinds of compatibility problems would have had to be considered. Such problems are beyond the scope of this chapter. However, they are very briefly touched on a the end of this chapter, when the concept of a *gateway* is introduced.

Objectives

The development of distributed database systems is driven by a set of objectives (i.e. what is to be achieved). The main objective (referred to as the "fundamental principle"), as stated by Date (Date 1995) is:

"To the user, a distributed system should look exactly like a NON-distributed system."

This statement is to be understood mainly with respect to data manipulation operations. Suppose DB1 and DB2 are equivalent database systems, DB1 is non-distributed and DB2 is distributed. The above principle states that if an operation may be performed on DB1, then an identical operation may be performed on DB2. As far as the data definition language is concerned, this principle cannot be applied completely. This is because the user will have to specify where (and how) the data is to be stored.

In accordance with the fundamental principle, Date (Date 1995) also identifies a set of twelve objectives for the development of distributed database systems. In this section, each of them will be briefly described. They are not mutually independent, nor are they exhaustive.

There still is a lot of work to be done before proper distributed database systems will be available. At present, there are DBMSs that provide some facilities for distributed databases. However, this support is limited and differs from one system to another. This is because, if we are to use Date's framework (objectives) in the analysis, different vendors associate different levels of significance to the twelve objectives.

Local autonomy

This objective states that each site of the distributed database system should be autonomous, i.e. it should work with its local data and local users as if the other sites did not exist. It can be rephrased as follows. Suppose A and B are *any* two sites in a distributed database system, then:

- B should not be able to prevent the execution of any operation on A nor should it be allowed to influence its result, provided this operation needs access only to the local resources of A;
- The success of a local operation on A should not depend on B (or any other sites);

Of a particular importance is the special case, often encountered in practice, when one of the sites, say A, is down. The other sites should be able to carry out their local operations as if nothing had happened to A.

This objective is not fully achievable. There are situations when a site has to "listen", has to be lead by another one. Such situations are presented in the next section. Therefore, the objective has to be rephrased as: local autonomy should be achieved to a maximum extent possible.

No reliance on a central site

There are two reasons for this

- if all the sites rely on a central site then this will constitute a bottleneck in the distributed system;
- if the central site goes down, the whole system goes down.

This objective is a special case of the above. However, it was stated separately, because, as opposed to the previous one, it can be achieved. So, in a distributed database system,

there will be no master site that has global knowledge about the system and decides how the other (slaves) sites interact.

However, as we shall see later, there are situations in which, for a period of time, a certain site has to play the role of a master. This does not mean that there is a central site, because this role is given to a site only for a brief period of time and is transferred from one site to another.

Continuous operation

This objective too can be regarded as a corollary of the first one. The distributed system should be such that if one or more of its sites go down the remaining parts continue to function so that the overall system is not totally paralysed. The system might be slower and access to certain data may be impossible. However, the system will continue to work.

An illustrative parallel can be made. On one hand, if a person serves in a shop and the person gets ill, the shop has to be temporarily closed, until the person gets well. On the other hand, if there are five shop assistants, then even if one of them gets ill, the shop will still continue to function.

This objective is reflected in an increased

- *reliability* – the users (application programs) find the system running when they need it (i.e. at different moments of time);
- *availability* – the system is continuously running for a particular period of time.

Location independence

Users should not have to know where the data is stored. They should be able to interact with the (whole) system as if the data were stored on their local site. This objective is often referred to as *location transparency*. Accordingly, data can move from one site to another without affecting the application programs (therefore this is a kind of program – data independence). Data might be moved from one site to another for efficiency reasons or when a site goes down.

Fragmentation independence

Before explaining fragmentation independence, the concept of *data fragmentation* must be introduced. If a system allows a relation to be split into parts (fragments) and stored accordingly, then the system supports data fragmentation.

Data fragmentation is purely motivated by performance reasons. The result of the logical design stage is a set of base relations in, say, at least BCNF. They represent a good logical structure in which data is to be stored (i.e. they minimise the possibility of update anomalies), but they might not represent the most appropriate form of storage in terms of efficiency of access.

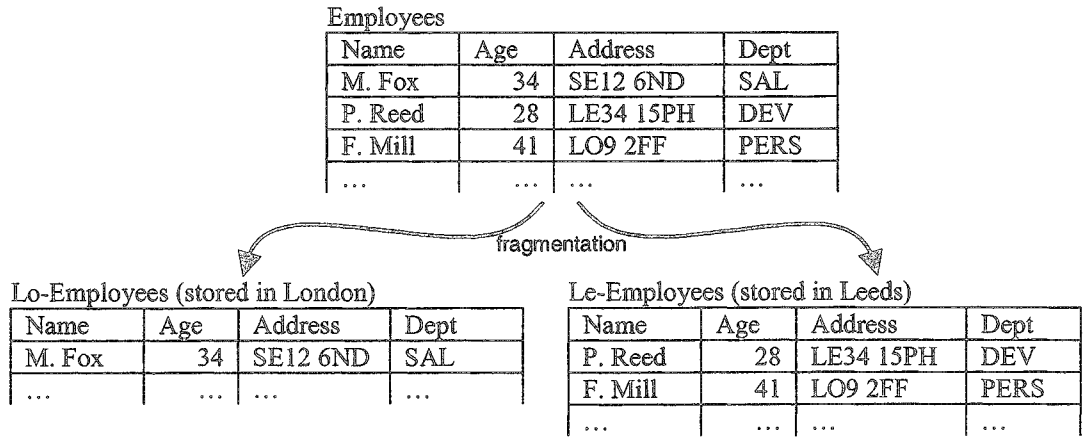

Figure 3: Example of a fragmentation

For instance, suppose that a company has the sales (SAL) department in London and the personnel (PERS) and development (DEV) departments in Leeds. Suppose, also, that, after normalisation, the information about the company's employees is to be stored in one relation, namely Employees. If this relation is to be stored in London, then it will be expensive for the branch in Leeds to access information about the employees who work

for the Personnel and Development departments (because of the network traffic). A similar situation occurs if the relation is stored in Leeds. The best solution is to split the relation (horizontally) into two; the part of the relation referring to the employees of the Sales department, Lo-Employees, will be stored in London and the part referring to the employees working for Development and Personnel, Le-Employees, will be stored in Leeds (Figure 3).

There are two basic types of data fragmentation:

- *horizontal*– performed by means of a *restriction* operator;
- *vertical* – performed by means of a *projection* operator.

These two can be combined to perform more complex data fragmentation. However, there are two restrictions that data fragmentation must comply with:

- all the fragments of a given relation have to be disjoint;
- for any vertical fragmentation the projection have to be non-loss.

The former restriction is to be understood as follows. If the fragmentation is horizontal, then the intersection of the resulting fragments has to be empty. If the fragmentation is vertical, then the only attributes that the resulting fragments should have in common should be those necessary to perform the join in order to re-establish the original relation.

For example, the horizontal fragmentation above is disjoint, because an employee only works for a single department. Therefore, Lo-Employees and Le-Employees have no tuple in common. As far as the vertical fragmentation is concerned, Figure 4 illustrates both a case of a disjoint and a non-disjoint fragmentation of a given relation.

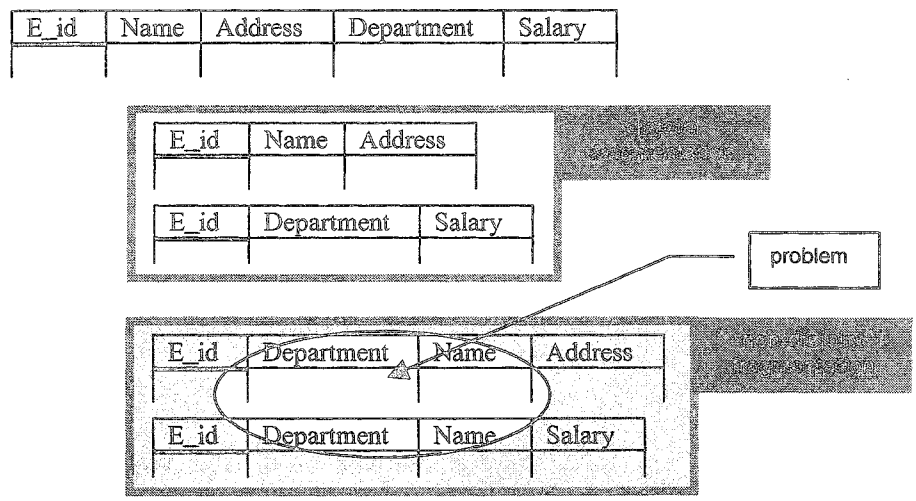

Figure 4: Examples of vertical fragmentation

The objective of *fragmentation independence* states that, users should perceive fragmented data, in their interaction with the system, as if it were not fragmented at all. This objective is also known as *fragmentation transparency*. This is also a kind of program-data independence. This means that data fragmentation can be dynamic, i.e. it can change in time, without affecting the application programs.

Since users always perceive the data as if it were not fragmented, it is the system's responsibility (more precisely of the *optimiser* of the system) to determine which fragments to use when issued with statements from the users.

For example, suppose that a database system, including the fragmentation of Figure 3, is issued with the following query (expressed in relational algebra):

```
Employees WHERE Dept = "DEV" AND Age < 40
```

The optimiser "knows" that the relation Employees was fragmented horizontally into two relations Lo-Employees and Le-Employees stored in London and Leeds,

respectively. Accordingly, Employees can be restored from the two by means of a UNION operation. Therefore, the optimiser transforms the above query into:

```
( Lo-Employees UNION Le-Employees ) WHERE Dept = "DEV" AND Age < 40
```

Distributing the WHERE clause, the above formula is transformed into:

```
( Lo-Employees WHERE Dept = "DEV" AND Age < 40 )
UNION
( Le- Employees WHERE Dept = "DEV" AND Age < 40 )
```

The optimiser knows the definition of Lo-Employees and Le-Employees, namely

```
Employees WHERE Dept = "SAL"
and
Employees WHERE Dept = "DEV" OR Dept = "PERS"
```

therefore it concludes that in order to evaluate the query, it only needs to use the fragment stored in Leeds (because Dept = "DEV" AND Dept = "SAL" can never be true).

We conclude by pointing out the similarity between the fragmentation and the view mechanisms.

Replication independence

The previous objective required fragments to be disjoint. In other words, it specified that no data redundancy should be introduced by fragmentation. This requirement was imposed in order to restrict the fragmentation process to its pure form, i.e. not mixed with other processes (such as controlled redundancy).

A distributed database system provides a mechanism for controlled (intentional) redundancy, namely *replication*. Replication means that a given relation (or fragment) can have copies stored (replicated) at other sites.

Providing replication means:

- *better performance* – the sites that hold replicas will access the respective data locally, removing the need to communicate (via the network) with the site that keeps the original copy;
- *better availability* – in order to access data, it is sufficient for one of the sites holding a replica of this data to be active.

The objective of *replication independence* stipulates that users, in interaction with the system, should perceive data as if it were not replicated at all. Users should not have to deal with details about which copy to use; this should be entirely the responsibility of the system. This means that copies can be created and destroyed dynamically without affecting the application programs. Therefore, this too, is a kind of program-data independence. It is also known as *replication transparency*.

Data replication presents an important disadvantage: when an update is made upon certain data then all the copies of that data must be updated as well.

Distributed query processing

A query, in a distributed database system, may involve relations (or fragments) stored at different sites. Therefore, certain data would have to be transferred from a site to another, across the communication network. This process is very slow (compared to the time of accessing data on a hard disk, for instance), substantially increasing the time for processing a query. Therefore, an important principle in query processing is to minimise the data transfer over the network. This principle results in two consequences:

- *set level operators* are preferred over tuple level operators (to transfer a set of tuples is faster than to transfer tuple by tuple)
- the *optimiser* becomes of paramount importance in a distributed database system; a good evaluation algorithm may bring the execution time of a query from hours to seconds (this aspect is illustrated in the next section).

Distributed transaction management

In a distributed database system, a transaction issued at one site, say S0, needing access to data not available locally, will have to involve other sites in its execution. The optimiser at site S0 determines which sites to access for the execution of the issued

transaction, and starts the corresponding processes at these sites[1]. A process that is executed at one site in behalf of a given transaction is called an *agent* (Date 1995). Therefore, a transaction in a distributed database system consists of a set of agents executed at different sites (Figure 5). Note that an agent, in turn, can create other agents at different sites.

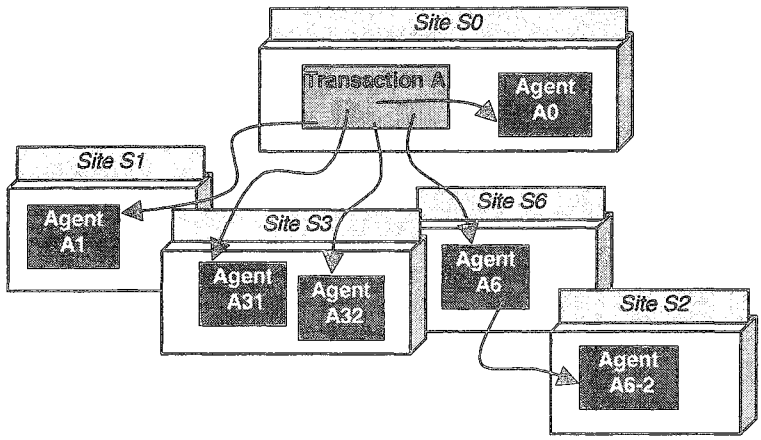

Figure 5: A transaction consists of a set of *agents*

As a result, each site will have a set of agents, belonging to different transactions. The system must have rigorous control over these agents in order to ensure that:

- the transaction that initiated a set of agents is atomic; i.e. either all the component agents are executed successfully or they are all rolled back (remember the two phase commit protocol);
- the concurrent access to data is correctly resolved (more particularly, if a locking mechanism is used, the system must ensure that no deadlocks will occur).

"System", above, means the overall distributed system and not a particular site. The control that the "system" exerts over the existing agents is achieved through its constituent sites: at different moments in time, different sites will control some of the other sites. This distribution – "who controls who" - changes in time.

The principle (objective) of local independence is violated by transaction management. However, since some control must exist in order to ensure transaction management, local independence cannot fully be achieved.

Hardware independence

The consequence of this objective is that different machines should be allowed to participate in a distributed database system.

Operating system independence

This principle is similar to the above, but it refers to the operating system rather than to the physical machine. They can be combined into a more general objective, namely *platform independence.*

Network independence

As above.

DBMS independence

This objective relaxes the homogeneity assumption, in that it specifies that it is desirable to have different collaborating DBMSs in a distributed database system, thus a heterogeneous system.

DBMS independence can be achieved by means of gateways (a special kind of interfaces, that make a DBMS look like another DBMS). For details refer to (Date 1995, pp. 614-616).

[1] This is in contradiction with the objective / principle of *local autonomy*. The site where the transaction was issues initiates processes at other sites, thus it *controls* the respective sites.

Problems

The previous section described what is expected from a distributed database system, i.e. the objectives that are to be achieved. This section looks at the main problems associated with the proposed objectives, i.e. what should it be resolved in order to accomplish them.

The sites in a distributed database system are linked via a wide area network (WAN). WANs are very slow for data transfer. Compared to the access to a disk drive, a WAN can be a thousand times slower. This characteristic has negative repercussions in important aspects of distributed database management.

There are two ways of alleviating this problem: (1) improve the data transfer time or (2) reduce the volume of data transfer over the network. The former objective undergoes substantial research, but there is still a long way to go until "reasonable" speeds will be achieved. Therefore, at least for the time being, it is more feasible to expect improvements in the performance of distributed database systems by focussing on the latter objective, i.e. minimising the utilisation of the network.

In the rest of this section we shall concentrate on some of the main problems encountered in developing distributed database systems.

Query processing

The way a query is evaluated (i.e. the order in which the corresponding operations are executed) in a distributed database system has enormous implications upon the processing time. Therefore, the *optimiser* (the module that decides how a query is to be evaluated) achieves a position of a paramount importance. A good optimiser can drastically reduce the execution time of a query. We shall illustrate this with an example (adapted from (Date 1995) pages 606 – 607).

Suppose that a distributed database system of a company includes information about the suppliers and the parts they supply. This is stored in three relations

- *Suppliers* – of attributes Supplier-Id and City (where the supplier is located);
- *Parts* – of attributes Part-Id and Colour;
- *Contracts* – of attributes Supplier-Id and Part-Id, describing which parts are supplied by which suppliers.

The Suppliers and Contracts relations are stored at site A and the Parts relation is stored at site B. Suppose the system is issued with the query

Get all the suppliers (Supplier-Id) from London who supply red parts.

This query is issued at a site C and can be expressed in SQL as

```
SELECT   Supplier-Id
FROM     Suppliers S, Contracts C, Parts P
WHERE    S.Supplier-Id = C.Supplier-Id AND C.Part-Id = P.Part-Id AND
         City = "London" AND Colour = "Red" ;
```

Suppose the following assumptions:

- every tuple of every relation is 200 bits long;
- Suppliers has 10,000 tuples, Parts has 100,000 tuples and Contracts has 1,000,000 tuples;
- the estimated number of red parts is 10;
- the estimated number of contracts with suppliers from London is 100,000;
- the data transfer rate is 50,000 bits per second;
- the access time (i.e., the time it takes to connect to a site in order to establish communication with) is 0.1 seconds;
- the computation time on each server (for calculations) is considered insignificant in comparison with the transfer / communication time.

We consider some possible evaluation strategies and compute the time it takes the query to be executed in each situation.

1. Move the Parts relation to site A and evaluate the result at A.
   This implies the following operations:
   - contact site A from site C and communicate evaluation method (0.1 seconds; the time taken to communicate the evaluation method is considered negligible, considering the small size of the message)
   - contact site B from site A (0.1 seconds)
   - transfer all tuples of Parts from B to A, i.e., 100,000 tuples (20,000,000 bits)
   - contact C from A (when the processing was completed) (0.1 seconds) and transfer the result to C (suppose the result has 100 tuples, i.e., 20,000 bits).

   The time that the first and last operations take, denoted by τ, is
   - $\tau = 0.1 + 0.1 + 20{,}000 / 50{,}000 = 0.6$ seconds

   This time occurs in all the other strategies. The time taken to complete the query is:
   - $\tau + 0.1 + 20{,}000{,}000 / 50{,}000 = 400.7$ seconds (approx. 6.678 minutes).
2. Move the Suppliers and Contracts relations to site B and evaluate the result at B.
   This implies the following operations:
   - contact B from C and communicate evaluation method (in τ);
   - contact A from B to ask for Suppliers (0.1 seconds)
   - transfer Suppliers from A to B, i.e., 10,000 tuples (2,000,000 bits)
   - contact A from B to ask for Contacts (0.1 seconds)
   - transfer Suppliers from A to B, i.e., 1,000,000 tuples (200,000,000 bits)
   - contact C from B when the processing was completed and transfer the result to C (in τ).

   The time taken to complete the query is therefore
   - $\tau + 0.1 + (2{,}000{,}000 + 200{,}000{,}000) / 50{,}000 = 4040.7$ seconds (approx. 67.345 minutes).
3. Restrict, at site A, the Suppliers relation to only those suppliers located in London, join the result with the Contracts relations (still at site A), project the result over {Supplier-Id, Part-Id} (so that the size of the tuple remains 200 bits) (still at site A), transfer the result to site B and evaluate the final result at B.
   This implies the following operations:
   - contact A from C and communicate evaluation method (in τ);
   - contact B from A, when the processing was finished at A, to communicate the rest of the evaluation method (0.1 seconds; the size of the message is negligible);
   - transfer the intermediate result from A to B, i.e. 100,000 tuples (still during the communication established above, so no other access time is needed);
   - contact C from B, when the processing was completed, and transfer the result to C (in τ).

   The time taken to complete the query is therefore
   - $\tau + 0.1 + 20{,}000{,}000 / 50{,}000 = 400.7$ seconds (approx. 6.678 minutes).
4. Restrict, at site B, the relation Parts to only red parts, transfer the result to A and finish the processing there.
   This implies the following operations:
   - contact B from C and communicate evaluation method (in τ);
   - contact A from B, when the processing was finished at B) to communicate the rest of the evaluation method (0.1 seconds; the size of the message is negligible);
   - transfer the intermediate result from B to A, i.e. 10 tuples (still during the communication established above, so no other access time is needed);
   - contact C from A, when the processing was completed, and transfer the result to C (in τ).

   The time taken to complete the query is therefore
   - $\tau + 0.1 + 2{,}000 / 50{,}000 = 0.74$ seconds.

Activity: Think of other evaluation strategies and compute the time it takes the query to complete.

Even for this simple example the execution time for the same query varies from a maximum of over one hour to a minimum of 0.74 seconds. More precisely, the fourth strategy is *5460 times* faster than the second strategy.

In the context of distributed database systems, there are two kinds of optimisation:

- *global optimisation* – in charge of deciding which sites to involve in the execution of the query and how to move data between sites, so that the best time is achieved; the example above illustrates global optimisation;
- *local optimisation* – in charge of deciding how to efficiently execute a local operation.

Developing a good optimiser is a very hard problem. The optimiser's decisions are based on assumptions, such as estimated intermediate results and speed of data transfer. Getting these assumptions right (i.e. estimated values as close as possible to the their real ones) is a difficult problem.

Catalogue management

The catalogue, in a distributed database system records not only information about base relations, views, users, etc., but also about locations of data objects, fragmentation and replication details. Such information is needed in order to provide location, fragmentation and replication independence.

The problem is where to store this catalogue. There are a few apparent possibilities. The catalogue can be stored:

- at a *single central site* – this severely violates the principle of no reliance on a central site, hence it brings all the drawbacks associated with it;
- *fully replicated* at each site – this is accompanied by a huge drawback, because each update to the catalogue must be propagated to all the other sites (therefore, it results in a loss of autonomy)
- *disjointedly partitioned* – each site maintains a catalogue for the objects it owns; since a site has no information about the data objects of the other sites, any non-local operation becomes very expensive.

None of the above possibilities represent a feasible solution. Certain combinations can be devised, but they would just place different emphases on the above drawbacks, they would not eliminate them.

In practice, different strategies are applied, that vary from a system to another. For instance, for the system R* (a prototype distributed database system built by IBM), the catalogue information about the location of the system's data objects is based on a naming convention. Each object has a name consisting of

- the creator of the object;
- the site where the object was created;
- the actual name of the object;
- the site where the object was initially stored.

This name is unique for every object and does not change during the life time of the object. Each site maintains the following catalogue information:

- an entry for every object born at that site (1);
- an entry for every object currently stored at that site (2).

When an object is to be accessed, the site where the object was created is determined from its name. This site is accessed and its catalogue looked up. If the object is still there, then an entry will exist in the catalogue (i.e., (2) above). If the object migrated to another site, then from (1) above, the location where it migrated to is identified. This example illustrates a possible approach to catalogue management.

Other approaches exist, but their presentation is beyond the scope of this chapter. The conclusion is important, namely that catalogue management is still a problem in distributed database systems.

Update propagation

Distributed database systems provide replication, i.e. the possibility of storing the copies of a data object at different sites. The advantages of replication were discussed in the previous section, so they are not mentioned here again. However, they are accompanied by a big drawback.

Every time an update is performed on a data object, this update has to be propagated to all its existing copies, in order to maintain the consistency of the database. Moreover, the initial update should not be allowed to be performed if any of the "propagated" update fail. However, some site that holds a copy of the data object might be down, making the update impossible. As a consequence, the principle of local autonomy is severely violated; a remote site does not allow for a valid update to be performed locally.

Replication was introduced to increase the availability of data, but, paradoxically, it can reduce it. A reasonable compromise solution is provided by the *primary copy scheme*.

- A copy of each data object is designated as a *primary copy*. All the others are considered *secondary copies*.
- An update operation is considered successfully completed at the moment when the primary copy was updated. It becomes the responsibility of the site holding the primary copy to propagate the update to all the other sites holding secondary copies. This can be done *after* the initial update was considered successfully completed. Two situations can occur.
  - The initial update is attempted on the primary copy. In this case, the site that hold the primary copy does not have to contact any other site; the update is performed completely locally.
  - The initial update is attempted on a secondary copy at site S1 (suppose site S0 hold the primary copy). In this case, S1 would have to communicate with S0 (but only with S0) and either perform the update together or not perform it at all.

This solution increases substantially the local autonomy but does not guarantee that the database will always be in a consistent state.

Recovery control

This topic was dealt with in the previous chapter, when the *two phase* protocol was introduced. This protocol is perfectly valid for distributed database systems too. Certain clarifications, however, must be made, in the light of the topics presented above.

- One site must play the role of the co-ordinator. This is usually performed by the site where the transaction was initiated. Therefore, all the other sites involved in the transaction must obey the instructions given by the co-ordinator. There are two consequences to this.
  - The two phase protocol implies the loss of local autonomy. The other sites must simply do what they are told by the co-ordinator.
  - However, the principle of no-reliance on a central site is not violated. There is no central site that controls the whole process of the system. Temporarily, a site S may become "central" from the point of view of some other sites, but after the transaction initiated at S is complete, S looses its "central" role. This role will be acquired by another site.
- The two phase protocol requires the co-ordinator to communicate with all the other sites (to enquire about the status of the operations they perform, to give instructions, etc.). Therefore the communication overhead could become quite substantial. However, there exists a variation of the basic two-phase commit protocol presented in the previous chapter, that reduce the number of messages needed.
- The two-phase protocol does not resolves any problem that might occur as a result of a failure; for certain situations it is possible that, if a failure occurs, the system will not be able to recover. As a matter of fact, no protocol exists that guarantees

that all agents commit together or roll back together, for any case of system failure (Date 1995, p. 612).

Concurrency control

Any request for a lock translates into messages, resulting in extra overheads. Therefore methods have to be found that reduce the number of messages. One of the possible solutions is provided by the primary copy approach previously described. For details refer to (Date 1995, p. 613).

Another problem is that the probability of falling into a deadlock is bigger. This is because, in a distributed system apart from local deadlocks, *global deadlocks* can also be generated. A global deadlock is a deadlock that involves two or more sites. An illustration is given by the diagram of Figure 6 (a global wait for graph).

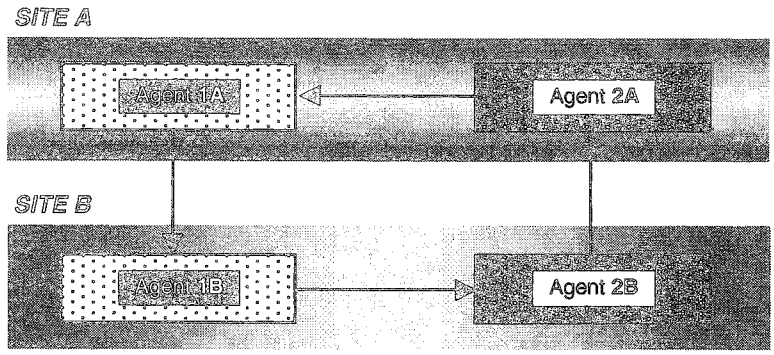

Figure 6: An example of a global deadlock

Transaction 1 has two agents, one on site A (1A) and one on site B (1B). Transaction 2 has two agents as well, one on A (2A) and one on B (2B). The situation is as follows. 1A is waiting for 1B, 1B is waiting for 2B, 2B is waiting for 2A and 2A is waiting for 1A to complete. Therefore, there is a deadlock. Such a deadlock cannot be identified by using only the local wait for graphs. Thus, a global wait for graph must be maintained, resulting into more overheads (messages).

Advantages and Disadvantages

This chapter introduced you to some of the most important objectives and some of the most relevant problems associated with distributed database systems. Based on them, the advantages and disadvantages of distributed database systems can be deduced. This is left as an exercise for you.

Activity: Identify some of the advantages and disadvantages of distributed database systems.

As a guidance, some of them are enumerated below. Advantages: matching the organisational structure, shareable but preserving local autonomy, improved availability, improved reliability, improved performance, modular growth. Disadvantages: complexity, cost, more difficult security and integrity control, lack of standards, lack of experience. For further help refer to (Connolly 1999, pp.652-654).

Learning outcomes

On completion of this chapter you should be able to:

- define the concept of distributed database systems;
- enumerate and explain the objectives of the development of distributed database systems;
- enumerate and explain the problems that occur in the development of distributed database systems.