CHAPTER

12

Distributed Database Management System

Contents

Learning Objectives

This chapter will enable the reader to understand the following:

- Need for distributed databases
- The difference among distributed databases, distributed systems, and parallel database systems
- Different types of DBMS architecture
- Concepts of data storage
- Processing of distributed databases and distributed query handling
- Database recovery in a distributed environment

12.1 Introduction

In the earlier chapters, we discussed various features of distributed systems such as their architecture, communication patterns, distributed shared memory, distributed file systems, etc. In this chapter, we will discuss the applications of distributed concepts, as implemented in database systems. Many new commercial database systems use distributed system concepts to get more cost-effective solutions to large databases. By definition, a distributed database is a logically interrelated collection of shared data (and its description) physically distributed over a computer network. Distributed databases have become economically feasible due to faster and more reliable communication networks and increased computing power.

The major reason for developing a database system is to integrate the operational data of an organization and provide controlled access to the data. A computer network promotes decentralization of data. The decentralized approach mirrors the organizational structure of many companies that are divided into departments, divisions, projects, etc. where each unit maintains its own operational data. The shareability of data and efficiency of data access should be improved by the development of a distributed database that reflects this organizational structure, makes the data in all units accessible, and stores the data proximate to the location where it is most frequently accessed.

As an example, consider the case of a nationalized bank like the State Bank of India. In Figure 12-1, we see the headquarters of State Bank of India in two different cities – Delhi and Mumbai.

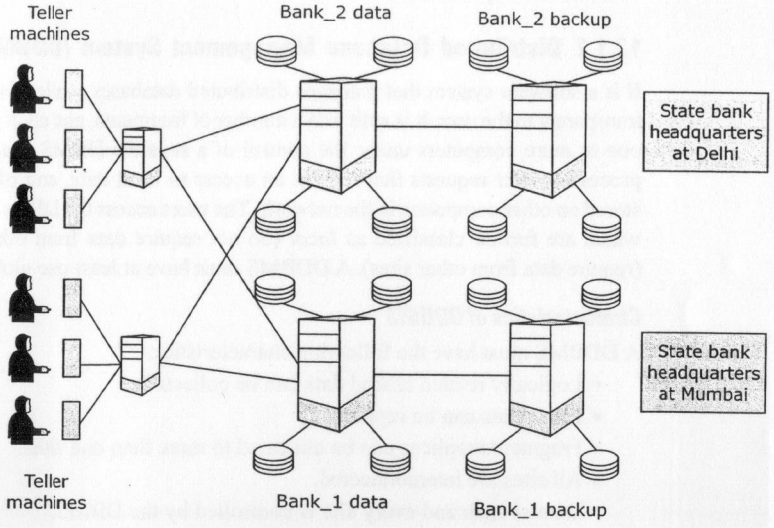


Figure 12-1 Nationalized bank's database

The whole setup is connected through a LAN within the city, and through the Internet across the country. The databases stored on the disks are replicated, to provide redundancy. A person having an account in Mumbai can access his/her account through an ATM at Delhi. The cash transaction is carried out as if his/her account is at Delhi. The entire transaction takes place in a database distributed across the Internet between the two main offices of the bank.

12.1.1 Distributed Database (DDB)

It is a collection of multiple, logically interrelated databases distributed over a computer network. In a DDB system, there are multiple computers called *sites* or *nodes*, which must be connected by some type of communication network, as shown in Figure 12-1. The databases are interconnected across a wide area network (WAN). They are maintained in duplicates, and each transaction is stored in both the databases. Thus, the databases form *islands of information*, separated by a geographical distance.

A DDB helps in resolving the 'islands of information' problem. The islands could be a result of geographical separation, incompatible computer architecture, incompatible computer protocols, etc. The logically related databases cooperate in a transparent manner, such that each user within the system may access all the data within all the databases as if they were a single database. There should be *location independence*, i.e. as the user is unaware of where the data is located, it should be possible to move the data from one physical location to another without affecting the user. Apart from the *database schema*, a distributed database system also consists of *fragmentation schema*, *allocation schema*, and *replication schema*.

12.1.2 Distributed Database Management System (DDBMS)

It is a software system that manages distributed databases while making the distribution transparent to the user. It is split into a number of fragments, and each fragment is stored on one or more computers under the control of a separate DBMS. Each site is capable of processing user requests that require an access to local data, and of processing the data stored on other computers in the network. The users access the DDBs through applications which are further classified as *local* (do not require data from other sites) and *global* (require data from other sites). A DDBMS must have at least one global application.

Characteristics of DDBMS

A DDBMS must have the following characteristics:

- Logically related shared data can be collected.
- Fragments can be replicated.
- Fragments/replicas can be allocated to more than one sites.
- All sites are interconnected.
- Data at each and every site is controlled by the DBMS.
- All local applications are handled by the DBMS at that site.
- Each DBMS takes part in at least one global application.

As shown in Figure 12-2, every site need not have its own local database. In the figure, there are four interconnected sites but only three databases to be shared within the network. A user can access any one of them from his login account on any one of the nodes. The database system should take care to conceal from the user the fact that he is accessing more than one database. This gives *distribution transparency*.

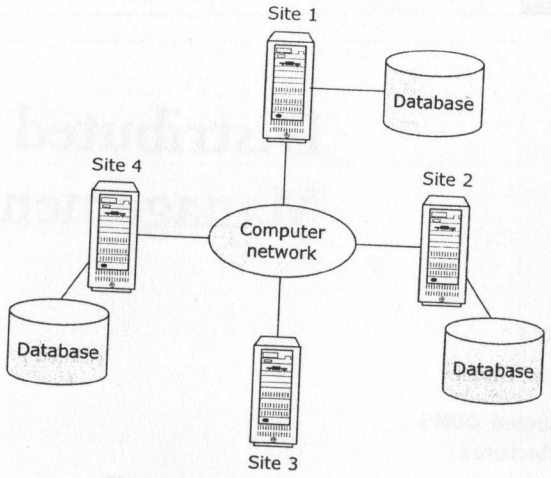


Figure 12-2 Distributed database

A DDBMS takes care of distribution transparency, replication transparency, and fragmentation transparency. It is hidden from the user that the distributed database is split into fragments, stored on different computers and replicated. This makes the DDBMS system appear like a centralized one to the users. In fact, transparency is the most fundamental principle of a DDBMS.

12.1.3 Distributed Processing

It is important to differentiate between a DDBMS and distributed processing. A distributed computing system, as shown in Figure 12-3, consists of a number of processing elements, not necessarily homogeneous, interconnected by a computer network. These processing elements cooperate in enabling access to centralized data. This can be seen in Figure 12-3 where a database stored at site 2 can be accessed from all sites.

A distributed database applies the concept of fragmenting a centralised data on multiple nodes and accessing them as if those were a homogeneous entity. Thus a DDBMS differs from distributed processing system. Typically, a database application distributes front-end presentation tasks to client computers and allows a back-end database server to manage shared access to a database. Consequently, a distributed database application processing system is more commonly referred to as a client/server database application

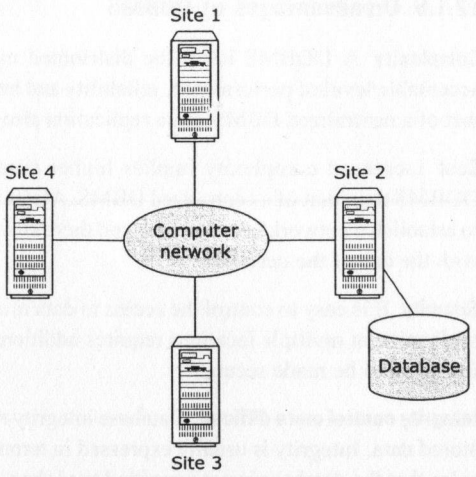


Figure 12-3 Distributed computing

system. Distributed database systems employ a distributed processing architecture. For example, an Oracle database server acts as a client when it requests data that another Oracle database server manages.

12.1.4 Parallel DBMS

We distinguish between a distributed DBMS and a parallel DBMS. A DBMS running across multiple processors and disks, designed to execute operations in parallel to improve performance, is a *parallel database*. A parallel DBMS is based on the premise that single processor systems can no longer meet the requirements for cost-effective scalability, reliability, and performance. A parallel DBMS links multiple, smaller machines to achieve the same throughput as a single, larger machine, with greater scalability and reliability than a single processor DBMS. To provide multiple processors with common access to a single database, a parallel DBMS must also provide for shared resource management. Performance and scalability are affected by the type of resources that are shared, and the manner in which the shared resources are implemented.

A parallel DBMS can have three types of architecture. They are:

- Shared-memory architecture
- Shared-disk architecture
- Shared-nothing architecture

The distribution of data in a parallel database is dependent on performance considerations. Typically, the nodes of a DDBMS are geographically distributed, separately administered, and may have a slower interconnection, while the nodes of a parallel DBMS are within the same site.

Shared-memory architecture It is a tightly coupled architecture with multiple processors in a single system sharing the system memory (see Figure 12-4). This is also called symmetric multiprocessing (SMP) and this approach is popular across workstations, Reduced Instruction Set Computers (RISC) based machines, and mainframes. It provides high-speed data access to a limited number of processors, i.e. the system is not scalable beyond 64 processors, past which point the interconnection network becomes a bottleneck.

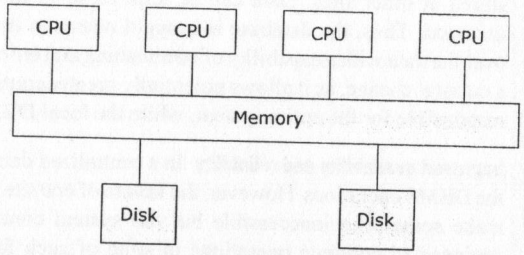


Figure 12-4 Shared-memory architecture

Shared-disk architecture It is a loosely coupled architecture optimized for inherently centralized applications that require high availability and performance. Each processor can access all disks but each has its own private memory. This architecture eliminates shared memory performance bottleneck. It also does not introduce any overhead associated with the physical partitioning of data. Shared disk systems are also called as *clusters*.

Shared-nothing architecture It is a multiprocessor architecture with each processor being a complete system having its own disk and storage. The database is partitioned among all disks on each system associated with the database and the data is available to users on all systems. This architecture is more scalable than shared memory, and can easily support large number of processors. Performance is optimal when the requested data is available locally. This architecture is also called as Massively Parallel Processing (MPP).

Parallel technology is suitable for large databases of the order of terabytes and for those systems that have to process thousands of transactions per second.

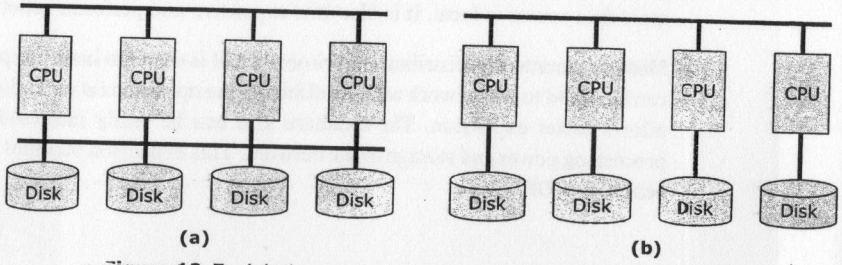


Figure 12-5 (a) shared-disk and (b) shared-nothing architecture

12.1.5 Advantages of DDBMS

Reflects organizational structure Many organizations are distributed over multiple geographical locations. Maintaining databases distributed over these locations gives frequent and faster local access.

Improved shareability and local autonomy The geographical distribution of the organization can be reflected in the distribution of the data, and a user at one site can access the data stored at other sites. Data can be kept close to the site from where it can be easily accessed. Thus, the database is brought nearer to its users and they have local control over the data with a capability of establishing and enforcing local policies. This can affect a cultural change, as it allows potentially greater control over local data. A global DBA is responsible for the entire system, while the local DBA manages the local data.

Improved availability and reliability In a centralized database, a computer failure terminates the DBMS operations. However, the failure of one site or a communication link failure may make some sites inaccessible but the system continues its operations. DDBMS are designed to continue operations in spite of such failures. In a DDBMS, data can be replicated and therefore, the failure of a node or a link does not make the data inaccessible. If a server fails, then the only part of the system that is affected is the relevant local site. The rest of the system remains functional and available.

Improved performance Since the data is located near the site of most frequent data access, and given the inherent parallelism of DDBMS, the speed of database access will be better than that of a centralized database. Since each site handles only a part of the entire database, there will be less contention for CPU and I/O services, as characterized by a centralized database.

Economics It costs less to create a system of smaller computers with the equivalent power of a single large computer. Thus, it is more cost effective for corporates and their departments to obtain separate computers or to add workstations to a network than to add a mainframe system. The second approach to cost saving is that, instead of one server handling the full database, we now have a collection of machines handling the same database. It becomes more economical to partition the application and perform the processing locally at each site. There is also a reduced communication overhead, since most data access is local. It is also less expensive and performs better.

Modular growth In a distributed environment, it is easier to handle expansion. New sites can be added to the network without affecting the operations at other sites. This flexibility allows easier expansion. The database size can be easily increased by adding more processing power and storage to the network. This expansion becomes very complex in a centralized DBMS.

12.1.6 Disadvantages of DDBMS

Complexity A DDBMS hides the distributed nature from the user and provides an acceptable level of performance, reliability and availability which is more complex than that of a centralized DBMS. Data replication also adds to the complexity.

Cost Increased complexity implies higher procurement and maintenance costs for DDBMS than that of a centralized DBMS. Also a DDBMS requires additional hardware to establish a network between sites and there are ongoing communication costs incurred with the use of the network.

Security It is easy to control the access to data in a centralized system, but in a DDBMS, replication at multiple locations requires additional control on access. Also the network itself has to be made secure.

Integrity control more difficult Database integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate. In a DDBMS, the communication and processing costs incurred to maintain integrity constraints is very large.

Lack of standards DDBMS depends on effective communication; and the lack of standards has significantly limited its potential. There are no tools or methods to help users convert a centralized DBMS into a DDBMS.

Lack of experience General purpose DDBMS have not been widely accepted and we do not have the same experience in industry as much as centralized DBMS.

Database design more complex As compared to a centralized DBMS, the design of a DDBMS has to take into account the fragmentation of data, the allocation of fragments to specific sites, and data replication.

12.1.7 Functions of DDBMS

A DDBMS software must be able to provide the following functions in addition to those of a centralized DBMS:

- Communication services to provide access to remote data
- Keeping track of data
- System catalog management
- Distributed query processing
- Distributed transaction management
- Replicated data management
- Distributed database recovery
- Security
- Distributed directory (catalog) management

At the physical hardware level,

- There are multiple computers called sites or nodes.
- These sites must be connected by some type of communication network to transmit data and commands among sites.

These functionalities will be discussed further in later parts of the chapter.

12.1.8 Types of Distributed Databases

A DDBMS is classified as *homogeneous* (all sites use the same DBMS product) or *heterogeneous* (sites may run different DBMS products). The system may be composed of relational, network, hierarchical, or object-oriented DBMSs.

Homogeneous DDBMS

In homogeneous systems, all sites have identical DBMS software. They are much easier to design and manage. This approach provides incremental growth, making the addition of a new site to the DDBMS easy and allows increased performance by exploiting the parallel processing capability of multiple sites. All sites are aware of one another and they agree to cooperate in processing users' requests. The software must also cooperate with other sites in exchange of information about transactions in order to make transaction processing possible across multiple sites. Figure 12-6 depicts a homogeneous distributed database system with a network of two or more Oracle databases that reside on one or more machines.

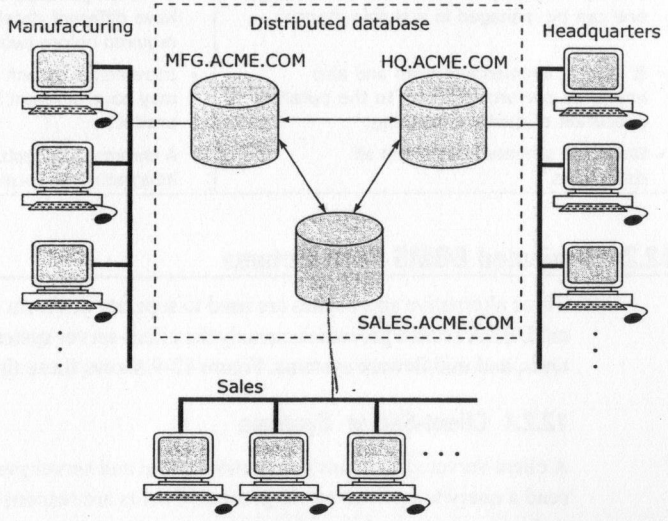


Figure 12-6 Homogeneous distributed database

Heterogeneous DDBMS

In heterogeneous systems, different sites may use different schemas and differe DBMS software, and the sites may not be aware of one another. This type of database designed when the sites have implemented their own databases and integration is consi ered later. To provide DBMS transparency, users must be able to make requests in t language of the DBMS at their local site. The sites may provide only limited facilities cooperate in transaction processing.

The system then has the task of locating the data and performing any necessary tran lation. Translations are required to allow communication between different DBMSs, a are also required to allow for different hardware, different DBMS products, or both. case the hardware is different, translation involves change of codes and word lengths. DBMS products are different, it involves the mapping of data structures in one model that in another model. If both hardware and software are different, then two translatio are required that makes the process more complex.

One typical solution is to use *gateways* that convert the language and the model each different DBMS into the language and model of the relational system. Gateways not support the problem of translating a query expressed in one language into transacti management, and this approach is only concerned with the problem of translating a que expressed in one language into an equivalent expression in another language. It does n address the issue of homogenizing the structural and representational differences b tween two different schemas. Figure 12-7 depicts the schematic view of a heterogeneo database.

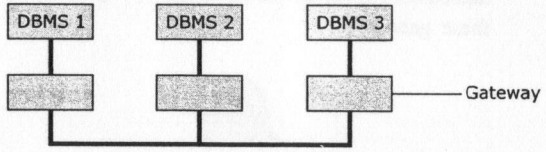


Figure 12-7 Heterogeneous database

Open database access and interoperability

The Open Group has formed a Working Group to provide specifications that will create database infrastructure environment where there is:

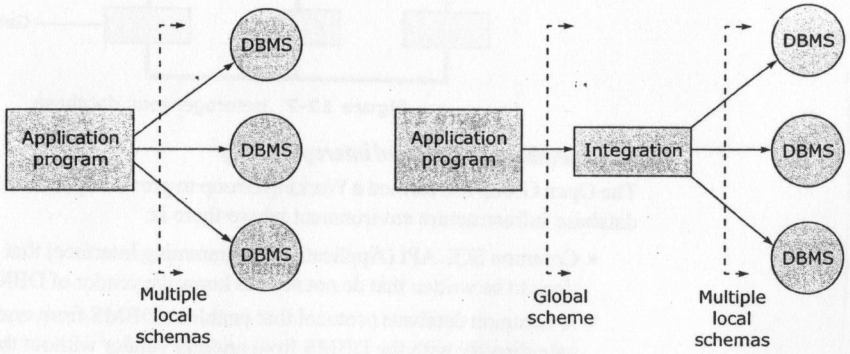
- Common SQL API (Application Programming Interface) that allows client applic tions to be written that do not need to know the vendor of DBMS they are accessin
- A common database protocol that enables a DBMS from one vendor to commun cate directly with the DBMS from another vendor without the need for a gatewa
- A common network protocol that allows communications between differe DBMSs.

The most ambitious goal is to find a way to enable transactions to span DBMSs fro different vendors without the use of a gateway.

Multidatabase systems (MDBS)

Recently, there has been a considerable interest in MDBSs, which logically integrate several independent DBMSs, and also allow the local DBMSs to maintain complete control of their operations. This complete autonomy implies that there can be no modifications to the local DBMSs. Also, manipulating the data residing in heterogeneous distributed databases requires an additional software layer on top of the existing database systems. The MDBS allows users to access and share data without requiring any physical database integration. However, it allows users to administer their own databases without a centralized control. The DBA of a local DBMS can authorize access to particular parts of a database by specifying an export schema that defines the parts of the database that can be accessed by non-local users. Figure 12-8 depicts multidatabase and integrated database schemas.

An MDBS is a DBMS that resides transparently on top of the existing database and file systems, and presents a single database to its users. The MDBS maintains a global schema against which users can issue queries and updates. The MDBS maintains only the global schema and the local DBMSs maintain all user data. The global schema is constructed by integrating the schemas of all local databases. The MDBS first translates global queries, updates them into local queries and updates the local DBMSs. Then it merges the local results and generates the final global result for the user. Next, the MDBS coordinates the commit and abort operations for global transactions by the local DBMSs that processed them, in order to maintain consistency of data within the local databases. An MDBS controls multiple gateways and manages local databases through these gateways.



(a) Multidatabase with local schemas

(b) Integrated distributed database with global schema

Figure 12-8 Multidatabase and integrated distributed database

An MDBS can be classified as:

- *Unfederated MDBS* (no local users)
- *Federated MDBS:* This is a cross between a DDBMS and a centralized DBMS. It is a distributed system for global users and a centralized system for local users. In other words, an MDBS is a DBMS residing transparently on top of existing database and file system that presents a single view to the user. The global schema is maintained by the MDBS and the data is managed by the local DBMSs. Users submit global queries and updates to MDBS. The global schema is integrated with the local schema and so the MDBS first translates the global queries and updates into local queries and updates of the appropriate DBMS. Finally, the MDBS merges the local results and presents the final global result. An MDBS also coordinates the commit and abort operations for global transactions by local DBMSs, necessary to have data consistency within the local databases. An MDBS controls multiple gateways and manages local DBMSs through them.

Table 12-1 compares the features of homogeneous and heterogeneous databases.

Table 12-1 Comparison of homogeneous and heterogeneous databases

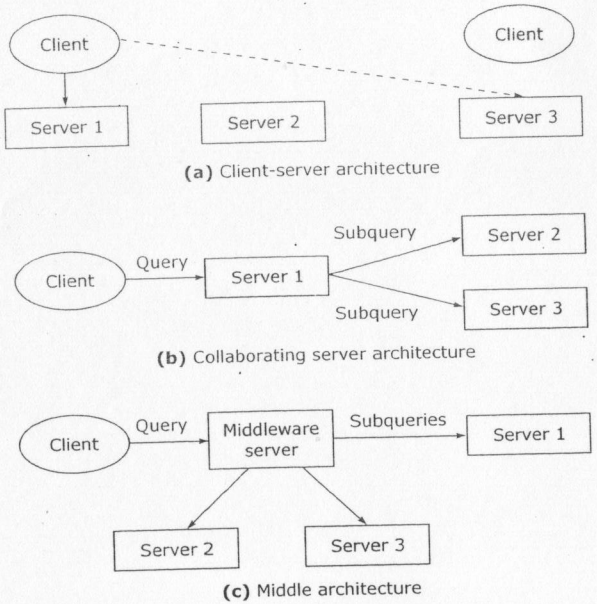
Homogeneous database	Heterogeneous database
• All sites use the same DBMS product.	• Sites may have different products with a different underlying model, such as network, relational, or object-oriented.
• Homogeneous systems are easier to design and can be managed in a simple manner.	• In heterogeneous systems, individual sites may have different databases and so translations are required before two databases can communicate.
• It allows incremental growth and also improved performance due to the parallel execution of query processing.	• Incremental growth may be difficult and a new site may have different hardware and different DBMS products.
• Database schema is same on all databases.	• A common conceptual schema has to be formed, integrating all local conceptual schemas.

12.2 Distributed DBMS Architectures

Three alternative approaches are used to separate different functionalities across different DBMS related processes, namely the client-server systems, collaborating server systems, and middleware systems. Figure 12-9 shows these three types of architectures.

12.2.1 Client-Server Systems

A client-server system has one or more client and server processes. A client process can send a query to any one server process. Clients are responsible for user interface issues and servers manage data and execute transactions. It is possible that a client process can run on a desktop and can send queries to a server running on a mainframe.



(a) Client-server architecture

(b) Collaborating server architecture

(c) Middle architecture

Figure 12-9 DDBMS architectures

It is one of the most popular architectures for the following reasons:

- It is simple to implement because the functionality is separated due to the centralized server.
- Expensive server machines are effectively utilized, since user interactions are handled by clients.
- Users can work with a familiar graphical user interface instead of using the interface on the server.

The communication between the client and server should be maintained as a set. Client-side caching can be used to reduce the communication overhead. One of the limitations of this architecture is that it does not allow a query to span multiple servers. This is because the client process does not have the capability to break the query into multiple subqueries, which can be executed at different sites and then put together. If it were to be made possible, the client process would need to be complex and end up doing many of the server-side tasks.

12.2.2 Collaborating Server Systems

The collaborating server system handles the limitation of the client-server architecture. It is a collection of database servers, each capable of running transactions against local data that cooperatively executes transactions spanning across multiple servers. When a server receives a query (which requires access to data at other servers), it generates specific

sub-queries that will be executed by other servers, and puts the results together to compute the answers to the original query. The decomposition of the query is done using cost-based optimization that is based on the cost of network communication and local processing costs.

12.2.3 Middleware Systems

The middleware architecture allows a single query to span multiple servers without requiring all database servers to be capable of managing multi-site execution strategies. This architecture is suitable for integrating several legacy systems whose basic functionalities cannot be extended. Only one database server is required for managing queries and transactions that span multiple severs, and the remaining servers handle only local queries and transactions. It is like having a special server—a layer of software that coordinates the execution of queries and transactions across one or more independent database servers. Such software is called a *middleware*; it is capable of executing joins and other relational operations on the data obtained from other servers, but does not itself maintain any data.

12.3 Data Storage in a Distributed DBMS

As discussed earlier, a DDBMS stores relations across several sites. Accessing data at a remote site incurs message passing costs. To reduce this overhead, a single relation may be either fragmented (or partitioned) across several sites. The fragments are stored at sites where they are often accessed or replicated at each site where the relation is in demand.

The definition and allocation of fragments are based on how the data is going to be used on the basis of the most frequently used transactions. The *quantitative information* is used in allocation, while the *qualitative information* is used in fragmentation. The quantitative information may include the *frequency* at which the transaction is run, the *site* from which it is run, and the *performance criteria* for the transactions. The qualitative information includes the information about the transactions which are executed such as relations, attributes, tuples accessed, type of access (read or write), and the predicates of read operation.

The definition and allocation of fragments are carried out strategically to achieve the following objectives:

- *Locality of reference:* As far as possible, the data should be stored closest to where it is used. In case a fragment is used at multiple sites, it may be advantageous to store copies of the fragments at these sites.
- *Improved reliability and availability:* Replication helps in improving reliability and availability, since there is a copy of the fragment available at another site, in case one site fails.
- *Acceptable performance:* Bad allocation may result in bottlenecks, since the site gets flooded with requests, causing a significant degradation in performance. This can also lead to underutilization of resources.

- *Balanced storage capacities and costs:* Using cheap mass storage can reduce costs and make the data available, but it has to be balanced against the locality of reference.
- *Minimal communication costs:* The costs of remote communication have to be taken into account. Retrieval costs are minimized when the locality of reference is maximized, or if each site has its own copy of data. In case a replicated data is updated, the update has to be performed at all sites holding a duplicate copy, thus increasing the communication costs.

12.3.1 Data Allocation

Replication is a part of data allocation that stores several copies of a relation, or a relation fragment. The relation or a relation fragment can be replicated at two or more, or even at all sites. The major motivations for replication are:

- *Increased availability of data:* If a site containing data crashes, the same data can be obtained from the site where the replica is kept. If the local copies of remote relations are available, the system is less vulnerable to failure of communication links.
- *Faster query evaluation:* Queries can execute faster by using a local copy of a relation, instead of getting the same from a remote site.

Basically, there are four alternative strategies regarding the placement of data:

- Centralized
- Partitioned (or Fragmented)
- Complete replication
- Selective replication

We compare the strategies based on the specific objectives described earlier.

Centralized The centralized strategy consists of a single database and a DBMS stored at one site, with users distributed across the network (referred to earlier as *distributed processing*). Locality of reference is lowest at all the sites except at the central site. Network access is required for all data accesses. Communication costs are high and reliability and availability are low, since a failure at the central site results in loss of entire database system.

Fragmented/partitioned This strategy partitions the database into disjointed fragments, and assigns each fragment to one site. If the data segments are located at the site where they are accessed frequently, the locality of reference is high. Storage costs are low, since there is no replication. Reliability and availability are also low, but are higher than the centralized allocation strategy. If the distribution is designed appropriately, then the performance is good and communication costs are low.

Complete replication It requires to maintain a complete copy of the database at each site. Hence, locality of reference, reliability and availability, and performance are maximized. However, the storage costs and communication costs for updates are expensive. *Snapshots* are used to overcome these problems. A *snapshot* is a copy of the data given at any time. The copies are updated periodically—hourly, daily, or weekly. Snapshots are also used to implement *views* in a distributed database to improve the time it takes to perform a database operation on a view.

Selective replication This strategy is a combination of partitioning, replication, and centralization. Some data items are *fragmented* to achieve high locality of reference, and those which are used at many sites and not frequently updated, are *replicated*, as otherwise the data items are *centralized*. The main objective of this strategy is to have all the advantages of all the approaches but none of the disadvantages of any of the approaches. This is the most common strategy, because of its flexibility.

Table 12-2 compares the features of all the strategies of data allocation.

Table 12-2 A comparison of data allocation strategies

	Locality of Reference	Reliability and Availability	Performance	Storage costs	Communication costs
Centralized	Lowest	Lowest	Unsatisfactory	Lowest	Highest
Fragmented	High	Low for item, high for system	Satisfactory	Lowest	Highest
Complete replication	Highest	Highest	Best for read	Highest	High for update, low for read
Selective replication	High	Low for item, high for system	Satisfactory	Average	Low

12.3.2 Fragmentation

As discussed earlier, the qualitative information is used in fragmentation. We first describe the major reasons of fragmentation.

- *Usage:* Applications are usually interested in 'views' and not the whole relations. Hence for data distribution, it is appropriate to work with the subsets of relations as the unit of distribution.
- *Efficiency:* It is more efficient if the data is close to where it is frequently used. Hence, the data is stored close to where it is most frequently used. Also, the data that is not needed by local applications is not stored.
- *Parallelism:* It is possible to run several sub-queries in tandem. With fragments as the units of distribution, a transaction can be divided into several sub-queries that operate on fragments. It increases the degree of concurrency in the system, and hence allows transactions that can be safely executed in parallel.
- *Security:* Data that is not required by local applications is not stored at the local site, and is not available to unauthorized users.

Fragmentation has two major *disadvantages*:

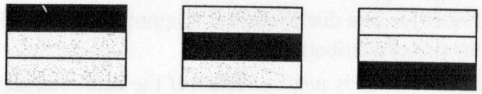
- *Performance:* The performance of global applications (which require data from several fragments located at different sites) is slower.
- *Integrity:* The integrity control is difficult if data and functional dependencies are fragmented and are located at different sites.

There are three *correctness rules* that need to be followed during fragmentation:

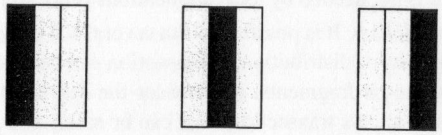
- *Completeness:* If a relation is decomposed into fragments, each data item must be found in at least one fragment. This rule is necessary to ensure that there is no loss of data during fragmentation.
- *Reconstruction:* It should be possible to define a relational operation which will reconstruct the relation from the fragments. This rule ensures that all functional dependencies are preserved.
- *Disjointness:* If a data item appears in one fragment, it must not appear in another fragment. *Vertical fragmentation* is an exception to this rule, where primary key attributes must be repeated in all reconstructions. This rule also ensures minimal data redundancy.

Fragmentation basically consists of breaking a relation into smaller relations of fragments, and storing the fragments preferrably at different sites. There are two types of fragmentation: (a) horizontal and (d) vertical.

In *horizontal fragmentation*, each fragment consists of a subset of rows of the original relation. Typically, the tuples belonging to a horizontal fragment are identified by a selection query. This storage achieves the locality of reference and reduces communication costs. In *vertical fragmentation*, each fragment consists of a subset of columns of the original relation. The tuples in a vertical fragment are identified by a projection query. When a relation is fragmented, we must be able to recover the original relation from the fragments. In case of horizontal fragmentation, the union of horizontal fragments must be equal to the original relation. Fragments are required to be disjoint. In vertical fragmentation, the collection of vertical fragments must be a *lossless join decomposition,* as per the definition. In general, a relation can be horizontally or vertically fragmented, and each of the resulting fragments can be further fragmented. This implies that they can be recursively partitioned. Figure 12-10 shows the types of fragmentation.



(a) Horizontal fragmentation



(b) Vertical fragmentation

Figure 12-10 Types of fragmentation

Let us consider a nationalized bank whose branches are in three main cities, Delhi, Chennai, and Mumbai. The employee database *B* has attributes as *eid*, *name*, *city*, *age*, and *salary*.

TID	Eid	Name	City	Age	Salary
T1	340001	Sunanda	Delhi	25	25000
T2	340002	Ramesh	Delhi	27	15000
T3	420003	Kalindi	Mumbai	30	34000
T4	420004	Kunal	Mumbai	32	52000
T5	430005	Kartik	Chennai	22	20000
T6	430007	Naresh	Chennai	24	22000

Here, the database can be fragmented along the *city* attribute and there can be three fragments. In horizontal fragmentation, all tuples in one city are grouped together. Thus, the database is partitioned into three parts, B_1, B_2, and B_3. When they are joined together, the database is fully reproduced.

The first partition (B_1) is as follows:

Eid	Name	City	Age	Salary
340001	Sunanda	Delhi	25	25000
340002	Ramesh	Delhi	27	15000

The second partition (B_2) is as follows:

Eid	Name	City	Age	Salary
420003	Kalindi	Mumbai	30	34000
420004	Kunal	Mumbai	32	52000

The third partition (B_3) is as follows:

Eid	Name	City	Age	Salary
430005	Kartik	Chennai	22	20000
430007	Naresh	Chennai	24	22000

Vertical fragmentation can be done with the grouping of certain attributes, for example, *TID*, *eid*, *name*, and *city* in one vertical fragment BV_1 and *TID*, *age*, and *salary* in another fragment BV_2. When both the fragments are joined together, they should have a lossless join, and so, another column *TID* that gives tuple ID, is added.

The first fragment (BV_1) is as follows:

TID	Eid	Name
T1	340001	Sunanda
T2	340002	Ramesh
T3	420003	Kalindi
T4	420004	Kunal
T5	430005	Kartik
T6	430007	Naresh

The second fragment (BV_2) is as follows:

TID	City	Age	Salary
T1	Delhi	25	25000
T2	Delhi	27	15000
T3	Mumbai	30	34000
T4	Mumbai	32	52000
T5	Chennai	22	20000
T6	Chennai	24	22000

Another possibility is *no fragmentation*. If the relation is small and not updated frequently, it may be better not to fragment it.

12.3.3 Replication

Replication is another technique in which some relations or their fragments are replicated and stored in multiple sites. An entire relation can be replicated in one or more sites.

Let us consider the above relation B which is fragmented into three parts B_1, B_2, and B_3, representing the fragments in the cities Delhi, Mumbai, and Chennai. Consider the fragment B_1 is replicated in sites 1 and 2; B_2 is not replicated; and B_3 is replicated in all sites.

Replication helps in having increased availability of data and also we can get faster query evaluation.

- *Increased availability of data:* As shown in Figure 12-11, if site 1 goes down, fragment B_1 is available at site 2, and can be restored from there. Similarly, B_3 also can be reconstructed from site 3.
- *Faster query evaluation:* Queries can execute faster if they use a local copy of data instead of taking the remote data.

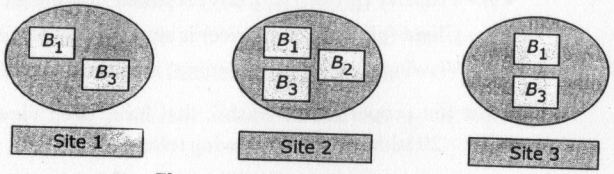


Figure 12-11 Data replication

The main problem with replicated data is maintaining its consistency. There are two types of replication: *synchronous* and *asynchronous*. Both differ in the technique of keeping the replicas current when any relation is modified.

12.4 Distributed Catalog Management

Replication of data at several sites is complicated. The system needs to keep track of how relations are fragmented and replicated. This implies how the relation fragments are distributed across several sites and where the copies of the fragments are stored including the schema, authorization, and statistical information. There are various approaches to catalog management, viz. centralized global catalog, replicated global catalog, dispersed catalog, and local-master catalog.

Centralized global catalog In this method, one site maintains the full global catalog. All changes to any local system catalog have to be propagated to the site maintaining the global catalog. This method gives bad performance, single point of failure, and compromises site autonomy.

Replicated global catalog In this method, each site maintains its own global catalog. Although it greatly speeds up remote data location, it is very inefficient to maintain. Any detail of every data item added, changed, or deleted locally has to be propagated to all other sites.

Dispersed catalog In this method, there is no physical global catalog. Each time a remote data item is required, the catalogues from all other sites are examined for the item. This has severe performance penalties.

Local-master catalog Each site maintains both its local system catalog as well as a catalog of all its data items that are replicated at other sites. It avoids compromising the site autonomy, is fairly efficient, and does not give a single point of failure.

12.4.1 Naming Objects

If a relation is fragmented and replicated, it should be possible to uniquely identify each replica of each fragment. A few requirements of this naming system are:

- Every data item must have a system-wide unique name.
- It should be possible to find the location of data items efficiently.
- It should be possible to change the location of data items transparently.
- Each site should be able to create new data items autonomously.

One of the methods is to use a global/centralized scheme (name server) to assign globally unique names. The name server assigns all names, with each site maintaining a record of local data items, and sites ask the name server to locate non-local data items. This method compromises the local autonomy—users at each site will not be able to

assign names to local objects without a reference to system-wide names. Also, the name server can become a potential performance bottleneck and/or a single point of failure.

One solution to the above problem is to use names consisting of several fields.

- *Local name field*: This is the name assigned locally at the site where the relation is created. Two objects created at two different sites can have the same local name, but two objects at the same site cannot have the same name.
- *Birth site field*: This field identifies the site where the relation was created, and where the information is maintained about all the fragments and replicas of the relation.

A combination of the above two fields identify a relation uniquely, and is called a *global relation name*. To find the replica, we take the *global relation name* and add a replica ID field. This combination is called the *global replica name*.

12.4.2 Catalog Structure

We have considered different approaches to distributed catalog management, viz. centralized global catalog, replicated global catalog, dispersed catalog, and local-master catalog, with varying catalog structure. If we consider a centralized catalog structure, it will be vulnerable to single site failure. A global systems catalog that describes all the data at every site may be used, but it compromises site autonomy because every change to a local catalog has to be broadcast to all sites.

The R distributed database project*, which was a successor to the *System-R Project* at IBM, followed a better approach to preserve local autonomy and avoid single site failure. It is as follows:

- Each site maintains a local catalog that describes all the copies of data stored at that site.
- The catalog at the birth site is responsible for keeping track of locations where the replicas and fragments of relations are stored.
- This catalog contains a precise description of each replica's contents, a list of columns for vertical fragments, or a selection condition for a horizontal fragment. Whenever a new replica is created or an old one is moved across sites, the birth site catalog must be updated.

For actually locating a relation, the catalog at the birth site must be looked up and that information cached at the local catalog. Whenever the relation has to be used again, the birth site catalog will be checked again to see if the relation has been moved. The birth site never changes even if the relation is moved.

12.4.3 Distributed Data Independence

Distributed data independence means that users must be able to write queries irrespective of how the relation is fragmented or replicated. The DBMS computes the relation as needed, by locating suitable copies of fragments, joining vertical fragments if required, and taking the union of horizontal fragments.

Distributed data independence also implies that users need not specify the full name for the data objects accessed while evaluating a query. As discussed in the previous section, the local name of a relation in the system catalog is a combination of a user name and a user-defined relation name. Users can give any name to the relation, without being bothered about the relations created by other users. When a user writes a program, he uses a relation name, while the DBMS adds a user name to the relation name to get a local name and then adds the user's site ID as the default birth site to obtain a global relation name. By looking up the global relation name in the local catalog (if it is cached) or in the catalog at the birth site, the DBMS can locate the replicas of the relation.

A user may want to create objects at more than one site or refer to relations created by other users. The user can create a *synonym* for the global relation name using any SQL-style command and then refer to it using this synonym. For each user known at a site, the DBMS maintains a table of synonyms as a part of the system catalog at that site, and uses this table to find the global relation name. The user's program remains unchanged even if the replicas are moved, since the global relation is never changed, unless it is destroyed. Users can run queries on specific replicas in case synchronous replication is used. Synonym mechanism can also be used to allow users to create synonyms for global replica names.

12.5 Distributed Query Processing

There are various issues involved in distributed query processing. A query may require data from more than one site. This data transmission involves communication costs. The data transfer costs for distributed query processing includes counting the number of page I/Os and the number of pages sent from one site to another because communication costs are a significant component of the overall cost in a DDBMS. If some of the query operations can be executed at the site of data, they may be carried out in parallel. Some of the operations like semi-join are used to reduce the size of a relation that needs to be transmitted and thus effectively economize on communication costs.

Let us consider an example of distributed query. A company has agents for selling its properties at different cities. They have maintained a database of clients, property and the numbers viewed. It is distributed in three cities, Delhi, Mumbai, and Pune. The three schemas and the size of relations are given below:

- p – Property (propertyno, city) is stored in Mumbai and has 10,000 records.
- c – Client (clientno, maxprice) is stored in Pune and has 100,000 records.
- v – Viewing (propertyno, clientno) is stored in Mumbai and has 1,000,000 records.

To list the properties in Nashik that have been viewed by the clients who have maxprice > 20 lakhs, we use following relation:

SELECT p.propertyno
FROM property p JOIN (Client C JOIN Viewing V ON C.clientno= V.clientno)
ON p.propertyno = v.propertyno
Where p.city = 'Nashik' AND C.maxprice > 2000000

Let us assume that each tuple in each relation is 100 characters long; that there are 10 clients with maxprice > 2000000; and there are 100000 viewings for properties in Nashik. Computation is negligible to communication costs, the data transmission rate is 10000 characters per second and there is a 1 second access delay. Table 12-3 gives the time spent in implementing different strategies.

Table 12-3 Strategies in distributed query processing

Strategy	Time
Move the client relation to Mumbai.	16.7 minutes
Move property and viewing relations to Pune.	28 hours
Join property and viewing at Mumbai. Select the tuples for Nashik and for each of these in turn, check at Mumbai if maxprice > 20 lakhs.	2.3 days
Select clients with maxprice > 20 lakhs at Pune and move the result to Mumbai for matching with the properties at Nashik.	1 second

The following equation is used to calculate the communication time.

$$\text{Time} = C_0 + (\text{message bits/transmission rate})$$

C_0 is called the access delay (the delay in setting up the communication).

Obviously, the last strategy is optimum. The data transfer is reduced considerably when the selection is done before transmitting a relation to Mumbai.

12.5.1 Non-join Queries in a DDBMS

A simple operation like scanning a relation, selection, and projection can be affected by fragmentation and replication. Let us consider the schema

BANKEMP B (empid, name, city, age, branchid, pay)

and the following query:

SELECT B.pay FROM BANKEMP B WHERE B.age > 18 AND B.age < 24

Suppose the relation BANKEMP is fragmented horizontally with all the tuples having age < 20 at Mumbai and age > 20 at Kolkata. The DBMS processes this query by evaluating it at both the sites and taking the union of the answers. If the SELECT clause contained AVG (B.pay), then the DBMS must compute the sum and count of pay values at both the sites, and use this information to compute the average pay of the employees. If the WHERE clause contained the condition B.age > 21, the query can be executed just at Kolkata.

Now let us consider vertical fragmentation. If the BANKEMP relation is vertically fragmented over *empid*, *name*, and *city* in one fragment and other attributes in another segment, the query processing would be done only at Kolkata. But it could be a *lossy join* unless another attribute TID (tuple ID) is replicated at both sites. Finally, if the fragments are replicated at both the sites, where will the query be processed? Suppose the answer is required at Mumbai, it will be better to process the query at Mumbai. The cost of shipping the answer to the query site and local processing cost are considered in deciding the site of query processing.

12.5.2 Joins in a DDBMS

Joins in any DBMS are very expensive, but it is more so in a distributed DBMS, where the fragments may be stored at distant sites, and unless properly constructed, a query may have a considerable communication overhead.

Let us consider the same example of property sale that we have described above. Here the relations *p* property and *v* viewing are stored in Mumbai and the relation *c* client is stored in Pune. It is assumed that the entire database is loaded in memory and there are no disk I/Os. We consider only the communication costs. All these points have been mentioned in the previous topic (see Table 12-3) but here we elaborate on them.

SELECT p.propertyno
FROM property p, INNER JOIN (Client C INNER JOIN Viewing V ON C.clientno= V.clientno)
ON p.propertyno = v.propertyno
Where p.city = 'Nashik' AND C.maxprice > 2000000

Let us consider the third strategy known as *Fetch as needed.*

Join *property* and *viewing* at Mumbai. Select the tuples for Nashik; and for each of these in turn, check at Mumbai if maxprice > 20 lakhs

We do a simple inner join where for each tuple in property, we join all the tuples in *viewing*. And get a relation of *property*, *city*, and *clientno* where only the tuples for Nashik city are selected. Then we check the client relation with the same clientno at Pune to see if the client is offering maxprice greater than Rs 2000000. The communication costs are computed as follows:

Each tuple is 100 characters long, there is an access delay of 1 second and a transfer rate of 10000 bits/second.

$$\text{Time} = 100000 \times (1 + 100/10000) + 100000 \times 1 = 2 \text{ or } 3 \text{ days}$$

This shows that *join* is a very costly operation and a simple join takes too much of communication load.

The other technique is *Ship to one site.*

Let us consider the first strategy. Since both the relations are in Mumbai, the Client relation should be brought to Mumbai and then the computation for time can be done. This involves transferring the entire Client relation from Pune to Mumbai and computing the number of clients with maxprice > 2000000.

$$\text{Time} = 1 + (100000 \times 100/10000) = 16.7 \text{ minutes}$$

As you see, the communication cost is considerably reduced. In this strategy, we have transferred 100000 client tuples to Mumbai and only 10 clients have maxprice > 2000000.

We should be able to optimize the query handling still further so that we can cut down the communications cost.

Semijoins and bloomjoins These techniques are used for reducing the number of tuples to be shipped from one site to another.

The first technique is called *semijoin*. We take a projection of clients with the condition of maxprice > 2000000 and transfer the reduced relation to Mumbai. At Mumbai, join the Client projection with the inner join of *property* and *viewing*.

The query becomes thus:

T1 [SELECT Clientno, maxprice from Client WHERE maxprice > 2000000]

SELECT p.propertyno FROM property p, T1, INNER JOIN (Client C INNER JOIN Viewing V ON C.clientno= V.clientno) WHERE p.city = Nashik AND T1.clientno = p.clientno

Relation *T*1 has only 10 tuples, since 10 clients satisfy the condition of maxprice and the transfer takes only 1 second, as shown in the calculation below.

$$\text{Time} = 1 + (10 \times 100/10000) = 1 \text{ second}$$

The technique of *bloomjoin* is similar to semijoin but instead of taking projection, a bit-vector of some size k is computed. The same hash function is used to hash join column values in the range 0–k. If some tuples hash to 1, the bit vector is set to 1; else to 0. This is repeated for both the relations and tuples hashing to 0 are discarded and only those corresponding to 1 are transferred to the other site.

Bloomjoin is more cost effective than semijoin but its effectiveness depends on the quality of hash function.

12.5.3 Cost-Based Query Optimization

Data distribution can affect the implementation of individual operations like selection, aggregation, join, etc. A query actually involves several operations and optimizing the queries in a distributed database poses a few challenges:

- Communication costs are incurred, because if we have several copies of a relation, we need to decide which copy to use.
- If individual sites are run under the control of different DBMSs, the autonomy of each site should be respected while doing a global query planning.

The query optimization process is very similar to that adopted in a centralized DBMS. The information about the relations at remote sites is obtained from the system catalogs. New options for distributed joins have to be considered and the cost metric should also account for communication costs.

In the overall plan, the local manipulation of relations at the site where they are stored is put into the suggested local plan. The overall plan includes various local plans which is like subqueries executing at different sites. To prepare the global plan, the suggested local plans provide a realistic cost estimate for the computation of intermediate relations. The suggested local plans are constructed by the optimizer to provide the local cost estimates. A site is free to ignore the local plan which is suggested, if it finds a cheaper plan by using more current information in the local catalogs. Thus, site autonomy is respected in the optimization and evaluation of distributed queries.

12.5.4 Updating Distributed Data

The traditional view of a DDBMS is that it should behave like a centralized DBMS from the user's point of view. Where updates are concerned, transactions should continue to be atomic actions, irrespective of data fragmentation and replication. There are two techniques to be followed regarding the updates:

1. Synchronous replication
2. Asynchronous replication

Figure 12-12 depicts the data updating techniques.

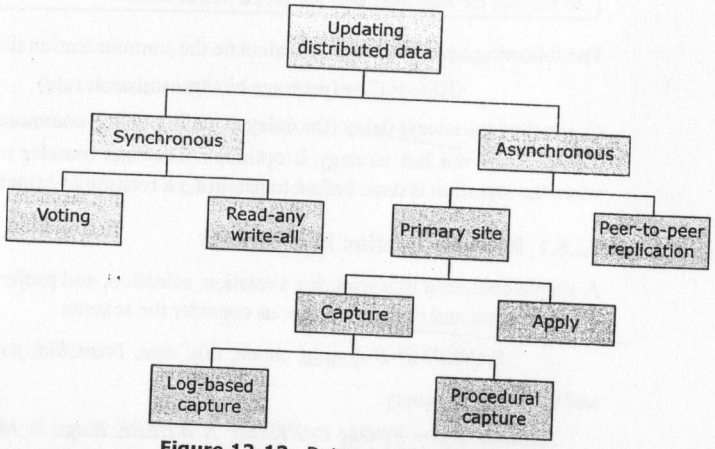


Figure 12-12 Data updating techniques

Synchronous replication

With respect to updates, a *view* means that the transactions must be atomic irrespective of data fragmentation and replication. All the copies of a modified relation (or fragment) must be updated before the modifying transaction commits. Replication with this semantics is called *synchronous replication*. Before an update transaction commits, it synchronizes all the copies of modified data. There are two basic techniques used for ensur-

ing that transactions see the same value irrespective of which copy of an object is accessed. They are *voting* and *read-any write-all* techniques.

Voting In this technique, a transaction must write a majority of copies to modify an object and read at least enough copies to ensure that one of the copies is current. For example, if there are at least nine copies and six are written by update transactions, at least four copies must be read. Each copy has a version number and the copy having the highest version is the current one. In this method, reading an object requires reading multiple copies, objects are read more frequently than they are updated, and efficient performance on reads is very important.

Read-any write-all In this technique, to read an object, a transaction can read any one copy; but to write an object, it must write all copies. *Reads* are fast, especially if we have a local copy but *writes* are slower as compared to the voting technique. This technique is advantageous when *reads* are more frequent than *writes*.

Synchronous replication comes with a significant cost. Before an update transaction can commit, it needs to obtain exclusive locks on all copies—if *read-any write-all* technique is used. The transaction will have to send lock requests to remote sites and will have to wait for the lock to be granted. In this case, there may be latencies and the transaction will continue to hold all other locks. In case the site or communication link fails, the transaction cannot commit until all the sites (at which it has modified data) recover and are reachable. If locks are obtained readily, and there are no failures, committing a transaction needs several additional messages to be sent as a part of the commit protocol. Hence, synchronous replication is undesirable in many situations.

Asynchronous replication

Since the cost for synchronous replication is generally very high, asynchronous replication is widely used in commercial DBMSs. The copies of a modified relation are updated only periodically, and it is possible that a transaction that reads different copies of the same relation sees different values. Hence, asynchronous replication compromises distributed data independence, but its implementation is more efficient as compared to synchronous replication.

This technique is gaining popularity, even though it allows different copies of the same object to have different values for short periods of time. This situation violates the principle of distributed data independence, requiring that the users must be aware of which copy they are accessing (keeping in mind that copies are updated only periodically) and work on a low level of data consistency.

Asynchronous replication can be categorized as: (a) Primary site replication and (b) Peer-to-peer replication.

Primary site replication In this type of replication, one copy of the relation is selected as the *primary* or *master copy*. This copy is updatable. Replicas, called *secondary copies*, can be created at other sites but they cannot be updated. One method of implementing

this mechanism is to first register/publish the relation at the primary site. Only then, it can be subscribed from any other site.

Let us consider how primary site replication can be implemented, so that the changes made to the primary copy are transferred to the secondary copies. There are two steps involved here: *capture* and *apply*. Whenever some changes are made to the committed transactions of the primary copy, they are identified during the *capture*, and then transmitted to secondary copies through the *apply* step. A transaction that modifies a replicated relation locks only the primary copy during the transaction processing, and then commits it.

Capture It can be either *log-based* or *procedural* capture. The log maintained for recovery is used to generate a Change Data Table (CDT). If this is done when the *log tail* is written to the stable storage, it should be modified to remove the changes of aborted transactions; and thus the CDT should contain only the committed transactions. In *procedural capture*, a procedure that is automatically invoked by the DBMS (a trigger) initiates the capture process, by taking a snapshot of the primary copy.

Log-based capture is the better option of the two, since it results in a smaller delay between the time the primary copy is changed to the time the change is propagated to the secondary copies. In particular, only changes and related changes are propagated. However, the drawback of this scheme is that the implementation of log-based capture is system-specific and requires a detailed understanding of the log structure. This belies the principle of transparency.

Apply The apply process at the secondary site collects the changes accumulated by the capture step and propagates them to the secondary copies. This can be done by *continuously* sending the changes done to the primary copy, or by doing so *periodically* based on a timer signal, or by a user's application program. In some cases, replicas can also be a view of the system defined by SQL, and the replication mechanism should maintain it incrementally.

Log-based capture with continuous apply minimizes the delay in updating the secondary replicas. This technique is best suited for situations when the primary and secondary copies are both used as a part of the operational DBMS, and the replicas are in close synchronization with the primary copy. Procedural capture and application-driven apply offer the most flexibility in processing the source data and changes before altering the replica.

Peer-to-peer replication In this type of replication, more than one copy are designated as master copies, and all copies are updatable. A conflict resolution strategy is incorporated to deal with the conflicting changes made at different sites. There are some typical situations where conflicts do not arise, where this method is suitable. Each master is allowed to update only a fragment of a relation; and two fragments updatable by different masters are disjoint. In another case, the updatable rights are held by only one master at a time. Changes at the master are then propagated to other sites.

12.6 Distributed Transactions

The objectives of distributed transaction processing in a DDBMS are more complex than that in a centralized DBMS, since they must also ensure the atomicity of global transaction and each component sub-transaction. The distributed database transactions should also have ACID properties that are necessary for all transactions. ACID stands for:

- *Atomicity*: A transaction is an indivisible unit that is either performed in its entirety or not at all.
- *Consistency:* A transaction must transform the database from one consistent state to another consistent state.
- *Isolation:* Transactions execute independently of one another. Incomplete transactions should not be visible to other transactions.
- *Durability:* The effect of a successfully completed (committed) transaction is permanently recorded in the database.

There are four high-level data modules that handle transactions pertaining to concurrency control and recovery, viz. *transaction manager*, *scheduler*, *buffer manager*, and the *recovery manager*. These exist in each local DBMS. The transaction manager coordinates transactions on behalf of application programs, communicating with the scheduler (the module responsible for concurrency control). The scheduler maximizes concurrency control without allowing concurrently executing transactions to interfere with one another, which would compromise database consistency. If a failure occurs during transaction, the recovery manager ensures that the database is restored to the state it was in before the start of the transaction. The recovery manager also restores the database to a consistent state, following a system failure. The buffer manager carries out efficient transfer of data between disk storage and main memory.

Additionally, a DDBMS also consists of a global transaction manager or a *transaction coordinator* (TC) at each site that coordinates the local and global transactions initiated at that site. The data communication component handles inter-site communication. Figure 12-13 shows the coordination of a distributed transaction.

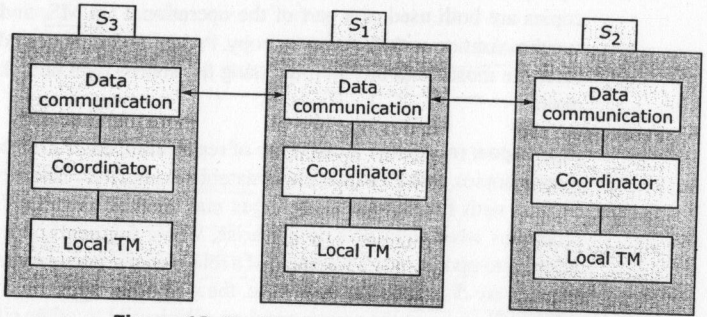


Figure 12-13 Distributed transaction coordination

As shown in Figure 12-13, a global transaction execution is initiated as follows:

- The TC at site S_1 divides the transaction into a number of sub-transactions using the information held in the global system catalog.
- The data communication component at S_1 sends the sub-transactions to the appropriate sites, for example, S_2 and S_3.
- The TCs at S_2 and S_3 manage these sub-transactions and the results are communicated back to TC_1 through the data communication components.

Based on this overview of a distributed transaction management, we next discuss the protocols for concurrency control, deadlock management, and recovery.

12.7 Distributed Concurrency Control

Concurrency control in a database is the process of managing simultaneous operations on the database without having them interfere with one another. In this section, we discuss the protocols for providing the concurrency control in a distributed DBMS. Let us first give some definitions and identify our objectives.

- *Schedule:* A sequence of the operations by a set of concurrent transactions that preserve the order of the operations in each of the individual transactions.
- *Serial schedule:* A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.
- *Non-serial schedule:* A schedule where the operations from a set of concurrent transactions are interleaved.
- *Locking:* A procedure used to control concurrent access to data. It allows only one transaction to access the database.
- *Shared lock:* If a transaction has a shared lock on a data item, it can read the item, but not update it. Other transactions can also read the item.
- *Exclusive lock:* If a transaction has an exclusive lock on a data item, it can read and update the item, but no other transaction can access that item.

12.7.1 Objectives

Assuming that the system has not failed, all concurrency mechanisms must ensure that data consistency is preserved, and each atomic action is completed in a finite time. The important capabilities of a good concurrency control mechanism for DDBMS are:

- be resilient to site and communication link failures
- allow parallelism to enhance performance requirements
- incur an optimal computational and storage cost
- perform satisfactorily in a network environment having communication delays
- place constraints on atomic actions

Various problems can arise when multiple users are allowed to access the database concurrently, like *lost update*, *uncommitted dependency*, *inconsistent analysis*, etc.

Similar problems exist in a DDBMS and there are additional problems that may arise due to data distribution. One of them is the *multiple consistency* problem which occurs when there are more than one copy of a data item at different locations. To maintain global consistency when a replicated data item is updated at one site, all the other copies must also be updated. If the copy is not updated, the data will be inconsistent. In this section, we assume that the updates to replicated items are carried out synchronously as a part of the enclosing transaction. The updates to replicated items can also be done asynchronously, that is, after the transaction updates the original coy of the data item that has been completed.

12.7.2 Distributed Serializability

The concept of serializability can be extended to distributed transactions to cater to data distribution in a distributed environment. If the schedule of transaction at each site is serializable, then the global schedule (union of local schedules) is also serializable, provided the local serializations are identical. This requires that all sub-transactions appear in the same order in the equivalent serial schedules at all sites. For example, if a sub-transaction of T_i at site S_1 is denoted by T_i^1, and if $T_i^1, < T_j^1$, then

$$T_i^x < T_j^x \text{ for all sites } S_x \text{ at which } T_i \text{ and } T_j \text{ have sub-transactions.}$$

The solutions to concurrency control in a distributed environment are based on two major approaches of *locking* and *timestamping*. A given set of transactions is said to be executed concurrently when:

- *locking* guarantees that the concurrent execution is nearly equal to some serial execution of those transactions.
- *timestamping* guarantees that the concurrent execution is equal to a specific serial execution of those transactions corresponding to the order of timestamps.

If the database is centralized or fragmented and also is replicated, there are multiple copies of each data item. All such transactions are either local or can be performed at a remote site. A locking based protocol can ensure that deadlocks do not occur. This involves checking for deadlocks at local and global levels by combining the deadlock data from more than one site.

12.7.3 Locking Protocols

In this section, we describe the various protocols based on two-phase locking (2PL) that can be used to ensure the serializability of a DDBMS:

1. Centralized 2PL
2. Primary copy 2PL
3. Distributed 2PL
4. Majority locking
5. Biased protocol
6. Quorum consensus protocol

Centralized 2PL

In this protocol, the system maintains a *single lock manager* that resides in a single chosen site, say S_i. There is a single lock manager/scheduler for the entire DDBMS that can grant and release locks.

The centralized 2PL protocol for a global transaction is initiated at site S_i and works as follows:

(a) The transaction coordinator at site S_i divides a transaction into several sub-transactions using the global system catalog. The coordinator ensures that consistency is maintained. If a transaction involves an update of a data item that is replicated, the coordinator will need to update all the copies of the data item. The coordinator requests exclusive locks on all copies before updating each copy, and before releasing the locks. The coordinator can elect to use any copy of the data item for a read operation.

(b) The local transaction managers that take part in the global transaction, request and release the locks from the centralized manager using the 2PL rules.

(c) The centralized lock manager now checks whether a request for a lock on a data item is compatible with the locks that currently exist. If so, the lock manager sends a message back to the requesting site, acknowledging that the lock is granted. Else, the lock is put in the queue till the lock request can be serviced.

A minor variation in this scheme is that the *transaction coordinator* makes all locking requests on behalf of the local transaction managers. Now, the lock manager interacts only with the transaction coordinator, instead of interacting with the individual local transaction managers.

The advantages of this scheme include its simple implementation, simple deadlock handling, and low communication costs. On the other hand, the lock manager site becomes a *bottleneck* and the system is *vulnerable* to lock manager site failures.

A global update operation with its agents (sub-transactions) at n sites may require a minimum of $2n + 3$ messages with a centralized lock manager (1 lock request, 1 lock grant messages, n update messages, n acknowledgements, and 1 unlock request).

Primary copy 2PL

The primary copy protocol overcomes the disadvantages of the centralized protocol by distributing the lock managers to many sites. Each lock manager is responsible for managing the locks for a set of data items. For each data item, one replica of data item is chosen to be the *primary copy,* while the other copies are marked as *slaves*. The site containing the replica is called the *primary site* for that data item. Different data items can have different primary sites. The site that is chosen to manage the lock for a primary copy, need not hold the primary copy of that item. When a transaction needs to lock a data item Q, it requests a lock at the primary site of Q. Then it implicitly gets a lock on all replicas of the data item.

Its main difference as compared to the centralized protocol is that to *update* a data item, the transaction coordinator must *locate* where the primary copy is. It is required to lock only the primary copy of the data item that is to be updated. Once the primary copy is updated, the changes are propagated to the slave copies. This is done immediately to prevent other sites reading out-of-date values. It is not necessary to carry out the updates as an atomic operation. This protocol only guarantees that the primary copy is current.

Advantages of the scheme include:

- Concurrency control for replicated data is handled similarly to unreplicated data. It is a simple implementation.
- Lower communication costs, and better performance than centralized systems.

Disadvantages of the scheme include:

- If the primary site of Q fails, Q becomes inaccessible, even though other sites containing a replica may be accessible.
- Deadlock handling is complex because of multiple lock managers, but there is some degree of centralization: lock requests for a specific primary copy are handled only by the primary site.
- This protocol is suitable when data is actively replicated, updates are infrequent and when sites may not always require the latest version of the data.

Distributed 2PL

This protocol attempts to overcome the disadvantages of the centralized 2PL by distributing the lock managers to every site. Each lock manager controls the access to local data items. If the data is not replicated, then this protocol is equivalent to the primary copy 2PL. However, special protocols may be used for replicas like the read-one-write-all (ROWA) replica control protocol.

As per this protocol, any copy of a replicated item can be used for a read operation but all copies must be exclusively locked before an item can be updated. The locks are dealt in a decentralized manner to avoid the drawbacks of centralized control.

The advantage is that the lock manager's work is distributed and can be made robust to failures. However, the disadvantage is that deadlock detection and handling is more complicated because of many lock managers. Also, the communication costs are higher than the primary copy 2PL, since all items are locked before update. With this protocol, a global update operation having its agents at n sites will require a minimum of $5n$ messages, i.e. n (lock request messages, lock grant messages, update messages, acknowledgements, and unlock requests). If the unlock requests are replaced by a single commit operation, the number of messages can be reduced to $4n$.

Majority protocol

The majority locking protocol is an extension of the distributed 2PL and it obviates the need to lock all the copies of a replicated item before an update. There is a local lock manager at each site to administer lock and unlock requests for the data items stored at

that site. When a transaction wishes to read or write a data item that is replicated at n sites, it sends a lock request to more than half of the n sites where the data item is stored. The transaction will proceed after it obtains locks on the majority of the copies. If the transaction does not receive a majority within a certain timeout period, it cancels the request and informs all sites that it has the lock. Any number of transactions can simultaneously hold a shared lock on a majority of copies but only one transaction can hold an exclusive lock on a majority of the copies.

This protocol avoids the drawbacks of centralized control. The other benefit is that it can be used even when some sites are unavailable. The disadvantage is that deadlock is more complex and locking requires at least $[(n + 1)/2]$ messages for lock requests and $[(n + 1)/2]$ massages for unlock requests. Correctness requires only a single data item to be locked (the item that is read) but this technique requests locks on all copies. There is potential for deadlock even with single item, for example, each of the three transactions may have locks on one-third of the replicas of a data.

Biased protocol

In biased protocol, the local lock manager at each site works as in the majority protocol. However, the requests for shared locks are favored more than the requests for exclusive locks.

- *Shared locks:* When a transaction needs to lock a data item Q, it simply requests a lock on Q from the lock manager at one site containing a replica of Q.
- *Exclusive locks:* When a transaction needs to lock a data item Q, it requests a lock on Q from the lock manager at all sites containing a replica of Q.

The biased scheme has the advantage of imposing less overhead on read operations than the majority protocol. This is more suited to applications having more reads than writes.

Quorum consensus protocol

This protocol offers a generalization of both majority and biased protocols. The algorithm works as follows:

- Each site is assigned a non-negative weight.
  - Let S be the total of all site weights
- Choose two values: read quorum (Q_r) and write quorum (Q_w)
  - Such that $Q_r + Q_w > S$ and $2Q_w > S$
  - Quorums can be chosen (and S computed) separately for each item

To execute a read operation, each read must lock enough replicas so that the sum of the site weights is greater than or equal to Q_r.

Similarly, to execute a write operation, each write must lock enough replicas so that the sum of the site weights is greater than or equal to Q_w.

The benefit of this approach is that it can permit the cost of either reads or writes to be selectively reduced by appropriately defining the read and write quorums. By setting the weights and quorums properly, the quorum consensus protocol can simulate the majority protocol and the biased protocol.

12.7.4 Timestamp Protocols

The objective of timestamping is to order a transaction globally such that older transactions (smaller timestamps) get priority in the event of a conflict. Timestamp-based concurrency-control protocols can be used in distributed systems too. Unique timestamps have to be generated locally and globally. Each transaction must be given a unique timestamp. The main problem is to generate a timestamp in a *distributed* fashion. Each site generates a unique local timestamp using either a logical counter or the local clock. The global unique timestamp is obtained by concatenating the unique local timestamp with the unique identifier (see Figure 12-14).

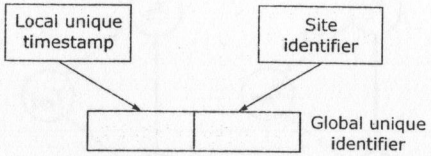


Figure 12-14 Timestamp management

A site with a slow clock will assign smaller timestamps that is still logically correct as long as serializability is not affected. However, the transactions with a faster clock will suffer as older transactions (those with lower timestamp values suffer most).

To fix this problem, the following algorithm can be used:

- Define within each site (S_i), a logical clock (LC_i) to generate the unique local timestamp.
- Require that S_i advance its logical clock whenever a request is received from a transaction T_i with a timestamp $<x,y>$ and x is greater than the current value of LC_i.
- In this case, site S_i advances its logical clock to the value $x+$, since a request can reach only after a delay.

12.7.5 Distributed Deadlock Management

Any locking-based concurrency control algorithm (and a few timestamping algorithms that require transactions to wait) may result in deadlocks.

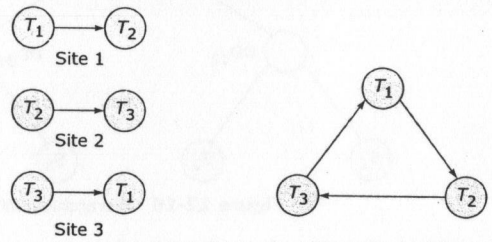
While handling concurrent transactions, we have to be careful to avoid deadlocks. Deadlocks can be prevented but the common practice is to detect them. Detecting a deadlock becomes more complex in case of distributed databases. Let us consider three transactions T_1, T_2, and T_3 initiated at three different sites S_1, S_2 and S_3. x, y, z are three data objects replicated at all three sites and referred as x_1 for the copy at S_1, y_2 for its copy at S_2, and z_3 at S_3.

- T_1 is initiated at site S_1 and creating an agent at S_2 to access the data item at S_2
- T_2 is initiated at site S_2 and creating an agent at S_3 to access the data item at S_3
- T_3 is initiated at site S_3 and creating an agent at S_1 to access the data item at S_1

Time	S_1	S_2	S_3
t_1	Read_lock (T_1, x_1)	Write_lock(T_2, y_2)	Read_lock (T_3, z_3)
t_2	Write_lock (T_1, y_1)	Write_lock (T_2, z_2)	
t_3	Write_lock (T_3, x_1)	Write_lock (T_1, y_2)	Write_lock (T_2, z_3)

Read locks are shared locks; and other transactions can access the data item x. Time is increasing downwards. At t_1, T_1 sets a shared lock on x, T_2 puts an exclusive lock on y, and T_3 puts a shared lock on z. At t_2, T_1 wants an exclusive lock on y but T_2 has already put an exclusive lock on y, so T_1 has to wait. Further at t_2, T_2 wants to put an exclusive lock on z but T_3 has put a shared lock on z. Another shared lock can be put but not an exclusive lock.

At time t_3, transaction T_3 wants an exclusive lock on x_1 but T_1 has put a shared lock on x_1 that should be released. Similarly at t_3, T_1 wants an exclusive lock on y but again T_2 has not released y. It is waiting for T_3 to release z. We can construct Wait-For-Graphs (WFGs) for all the three transactions locally and globally. As shown in Figure 12-15, there is no deadlock individually but if we combine the WFGs, we can see that a cycle exists.



(a) Local WFGs **(b)** Global WFG

Figure 12-15 Wait-for graphs

Figure 12-15 shows that local wait-for graphs are not sufficient to check the existence of deadlocks in a distributed system. A global graph that is a union of all WFGs must be constructed to look for deadlocks. One of the three methods described below is used for resolving the deadlocks: *centralized*, *hierarchical*, and *distributed*.

Let us consider one unique situation related to distributed databases only called the *phantom deadlock*. This is caused by the delays in propagating local information. Suppose that transaction T_1 is waiting for a resource locked by transaction T_2. T_2 has already released the resource but the message has been delayed and T_1 is unnecessarily waiting. So this leads to a false or phantom deadlock.

Centralized deadlock detection

A single site is appointed as the *deadlock detection coordinator* (DDC) that is responsible for constructing and maintaining the global WFG. Periodically, each lock manager sends its local WFG to the DDC. The DDC builds the WFG and checks for cycles. If one

or more cycles are detected, the DDC must break each cycle by selecting the transactions to be rolled back and restarted after sending a multicast to all sites involved in the processing of these transactions. Data transfer can be minimized by the lock manager by just sending the changes made to the local WFG since the last *send*. As is the case with all centralized algorithms, reliability suffers if the central site breaks down.

Hierarchical deadlock detection

For hierarchical deadlock detection, the sites in the network are arranged in a network. Each site sends its local WFG to the detection site above it, and finally the root of the tree constructs the global deadlock WFG. At each site, the deadlock manager checks for the existence of cycles and breaks them if any exist. Figure 12-16 shows the hierarchy of four sites, S_1 to S_4. Level 1 leaves are sites where the deadlock is checked locally. Level 2 nodes are DD_{ij} where deadlocks involving adjacent nodes are performed and so on. At the root level, the deadlock between S_1 and S_4 is performed.

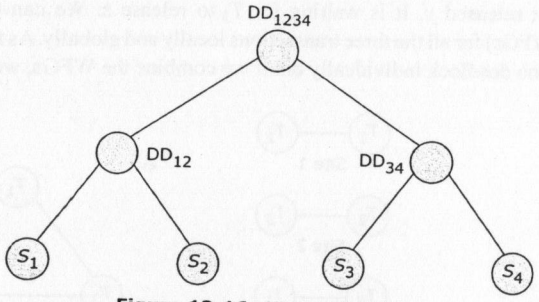


Figure 12-16 Hierarchical WFG

Hierarchical approach reduces the dependence on a centralized detection server and communication costs too, but is more complex and is prone to network failures.

Distributed deadlock detection

There are many different versions of this approach but the most common is that of adding an external node T_{ext} to the local WFG to hint that an agent is introduced at a remote site. For example, the global WFG shown in Figure 12-15 would be represented by the local WFGs at sites S_1, S_2, and S_3 as shown in Figure 12-17. The edges in the local WFG linking agent to the T_{ext} are labelled with the site involved. For example, the edge connecting T_1 and T_{ext} is labelled S_2, as this edge represents an agent created by the transaction T_1 at site S_2.

If a local site contains a cycle that does not involve the the T_{ext} node, then the site and the DDBMSs are in a deadlock. If the local WFG has a cycle involving the T_{ext} node, then there might exist a deadlock. Since T_{ext} may represent different agents, there may not be a global deadlock. So it is necessary to merge all WFGs and get a global picture. The graph then will be of the form:

$$T_{ext} — T_i — T_j — T_k — T_{ext}$$

Timestamp allocation should be done to prevent the sites from transmitting their WFGs to each other. A simple strategy works: allocate a timestamp to each transaction, let a site S_1 transmit its WFG to a site for which T_k is waiting, S_k, only if $ts(T_i) < ts(T_k)$. Then to check for the deadlock, site S_1 would transmit its local WFG to S_k. Site S_k would add this to its WFG and check for cycles not involving T_{ext} in the extended graph. If there is no such cycle, then the process continues till either a cycle appears (in which case, one or more transactions are rolled back and restarted together with all their agents), or the entire global WFG is constructed (in which case, there is no cycle). In such a case, there will be no deadlock in the system.

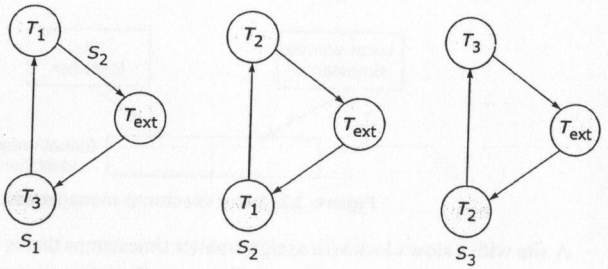


Figure 12-17 Distributed deadlock detection

The three local WFGs in Figure 12-17 contain cycles.

S_1: $T_{ext} — T_3 — T_1 — T_{ext}$

S_2: $T_{ext} — T_1 — T_2 — T_{ext}$

S_3: $T_{ext} — T_2 — T_3 — T_{ext}$

In Figure 12-17, we can transmit WFG at S_1 to the site S_2 for which T_1 is waiting. The local WFG is extended to include this information and then we get:

S_2: $T_{ext} — T_3 — T_1 — T_2 — T_{ext}$

It still contains a potential deadlock condition. So we can transmit this WFG to the site for which T_2 is waiting, i.e. S_3. The local WFG at S_3 becomes

S_3: $T_{ext} — T_3 — T_1 — T_2 — T_3 — T_{ext}$

This global WFG now contains a cycle even without involving T_{ext} and so we can say that a global deadlock exists, and relevant recovery should be invoked. Distributed deadlock detections are more robust but a lot of inter-site communication may be involved.

12.8 Distributed Database Recovery

Here, we discuss the protocols required for handling failures in a distributed environment. There are two new issues in recovering from failures. The first issue is the failure of communication links and remote sites, and the second issue occurs due to the possibility

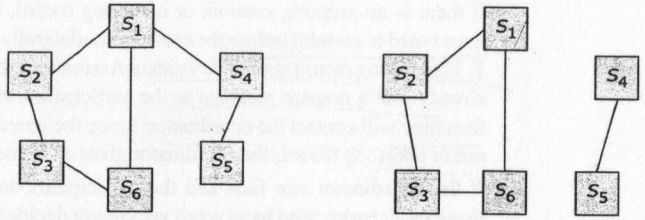
of a transaction divided into many sub-transactions and executing those sub-transactions at different sites. So the commit procedures are carried out in two or three phases.

12.8.1 Failures in Distributed Environment

The following types of failures are unique to distributed systems:

- Loss of message
- Failure of a communication link
- Failure of a site
- Network partitioning

The loss of messages or improperly ordered messages is the responsibility of the underlying network transmission control protocols, for example, TCP-IP. We assume that they are handled transparently by the communication component of the DDBMS. Failure of a communication link is another possibility. A DDBMS is dependent on the ability of all sites in the network to communicate reliably with one another. Currently, the network technologies are more reliable. However, failures can still occur. Communication failures result in the network being split into two or more partitions where the sites within the partitions can communicate with one another but not with the sites in other partitions.



(a) Network partition before failure

(b) Network partition after the failure of link S_1–S_4

Figure 12-18 Network partitioning

Figure 12-18 shows a network of six nodes. Actually it is difficult to distinguish whether a communication link has failed or the site has crashed. Suppose the node S_1 cannot communicate with S_4. Any one of the following reasons may exist:

- The site has crashed or the network is gone.
- The communication link has failed.
- The network is partitioned.
- Site S_4 is very busy and cannot respond.

A proper timeout value has to be chosen so that the exact picture of the failure becomes clear.

12.8.2 How Failures Affect Recovery

Similar to local recovery, distributed recovery aims to maintain atomicity and the durability of distributed transactions. To warrant atomicity of global transactions, DDBMS must ensure that sub-transactions of the global transaction must all commit or all abort. If the DDBMS detects that a site has failed or has become inaccessible, it needs to carry out the following steps:

- Abort any transactions that are affected by the failure.
- Flag the site as failed, to prevent other sites from using the failed site.
- Carry out periodic checks to see whether the site has recovered, or wait for the failed site to broadcast that it has recovered.
- On restart, the failed site must initiate a recovery procedure to abort any partial transactions which were active at the time of failure.
- After the local recovery, the failed site must update its copy of the database to ensure consistency.

In case a network partition occurs, the DDBMS must take proper care before committing transactions so that the global atomicity is not violated.

Distributed recovery protocol Recovery in a DDBMS becomes complicated by the fact that both global and local transactions should maintain their atomicity. In simple words, it means that in a DDBMS, a transaction is executed in multiple sites after being broken into sub-transactions. If all sub-transactions have successfully completed the transaction, then the entire transaction can be committed. Even one sub-transaction failing will lead to aborting all the transactions. This involves modifying the commit and abort protocols. We will discuss two most common commit protocols: *two-phase commit* and *three-phase commit*.

12.8.3 Two-Phase Commit (2PC)

The basic idea is that every transaction has one node (where the transaction is originated) that acts as the *coordinator* and all other nodes handling the sub-transactions act as *participants/subordinates*. 2PC operates in two phases—a *voting phase* and a *decision phase*. In the voting phase, the coordinator asks the subordinates whether they are ready to commit, and waits for their answers. If all of them say yes, then in the second phase, the coordinator gives the message to commit. If any one of them is aborting, then the coordinator sends the abort message to all subordinates. This is known as *unilateral abort*. If a subordinate votes to commit, it must wait for the coordinator to broadcast either a *global commit* or a *global abort* message.

Phase 1 includes the following steps:

- The coordinator sends a *prepare* message to each participant.
- Participants respond with a *yes/no*.

Phase 2 includes the following steps:

- If the coordinator receives all *yes*, then it sends the message *commit*; else *abort*.
- Each participant must *acknowledge* the commit or abort message.
- The coordinator on receiving an *acknowledge* message from all participants, writes an end log record.

Let us consider the two-phase commit in detail. There are two rounds of messages:

1. Voting round
2. Termination round

Let us assume the *fail-stop model* – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites. Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached. The protocol involves all the local sites at which the transaction is executed. Let T be a transaction initiated at site S_i and let the transaction coordinator at S_i be C_i.

Phase 1: Voting

- Coordinator C_i asks all participants to *prepare* to commit a transaction T_i.
  - C_i adds the records <*prepare T*> to the log and forces the log to stable storage
  - Sends *prepare T* messages to all sites at which T executed
- Upon receiving the message, the transaction manager at participant's site determines if it can commit the transaction.
  - If not, add a record <*no T*> to the log and send *abort T* message to C_i
  - If the transaction can be committed, then:
  - Add the record <*ready T*> to the log
  - Force *all records* for T to stable storage
  - Send *ready T* message to C_i

Phase 2: Termination

- T can be committed if C_i received a *ready T* message from all the participating sites, otherwise T must be aborted.
- Coordinator adds a decision record, <*commit T*> or <*abort T*>, to the log and forces the record onto stable storage. Once the record is in stable storage, it is irrevocable (even if failures occur).
- Coordinator sends a message to each participant to broadcast the decision (commit or abort).
- Participants take appropriate actions locally.

The two-phase commit exchanges the two phases of messages (voting phase and termination phase) between the coordinator and the participants. When a message is sent in 2PC, it means that the sender has taken a decision, and the log record describing this decision is forced to stable storage before the message is sent.

A transaction is officially committed when the coordinator's commit log reaches the stable storage. The log records of commit protocol contain the type of records, transaction-IDs, and coordinator-IDs. A coordinator's commit or abort log record contains the participant-IDs.

Restart after the failure When a site comes up after a failure, a recovery procedure is started. That procedure will process all transactions as per the log. The transaction manager can be a coordinator for some transactions and a participant for others. The recovery process is as follows:

- If there is a commit or an abort log record for the transaction T, then redo or undo the transaction. Determine from the log record if this site is the coordinator or a participant. If it is the coordinator, then periodically keep sending commit or abort messages to participants till an acknowledgement message is received.
- If there is a prepare log record for the transaction but no commit or abort record, then this site is a participant and the coordinator can be determined from the log record. Then repeatedly contact the coordinator to find the status of transaction and then *write*, *commit*, or *abort* the record, *redo* or *undo* the transaction, as the message is either commit or abort. Then write an *end* record.
- If there is no *prepare*, *commit*, or *abort* log record, then the transaction could not have voted to commit before the crash. So unilaterally abort and undo the transaction T. The *end* log record should be written. Assuming this site is the coordinator and has already sent a prepare message to the participants, and they might have voted yes, then they will contact the coordinator. Since the coordinator does not have any commit or abort log record, the coordinator gives a unilaterally abort message.
- If the coordinator site fails and the participants do not have any message, then those participants who have voted yes cannot decide whether to commit or abort till the coordinator recovers. Then the transaction T is blocked. Even if all participants know each other, they are blocked, unless one of them has voted no. In that case, they can abort the transaction.
- If a remote site does not respond during the commit protocol for the transaction T, then may be the communication link has failed or the site has failed. Then one of the following activities may take place:
  - If the current site is the coordinator for T, it should abort T.
  - If the current site is a participant and has not voted yes, then it should abort T.
  - If the current site is a participant and has voted yes, then it is blocked till the coordinator responds.

2PC with presumed abort Consider three basic observations regarding 2PC protocols:

1. The acknowledgement messages in 2PC are useful in knowing whether all participants are aware of the decision (commit or abort). Till then, the coordinator must keep the information about the transaction in the transaction table.

2. The coordinator site fails after sending the *prepare* message, but before writing a *commit* or *abort* log record. In that case, it has no information about T after it comes up. Hence, it is free to abort and in the absence of that information, it is supposed to have aborted.
3. If a sub-transaction does no updates, it has no changes either to redo or undo. So it need not take any action. It is just a reader.

Refinements for 2PC The first two observations show that 2PC can be refined further, as follows:

- When the coordinator aborts a transaction T, it can undo T and remove it from the transaction table immediately, so that the default response is abort.
- If a participant receives an abort message, then the acknowledgement message is not needed, because the coordinator removes the information about T from the transaction table. If participants do not receive any commit or abort message after the *prepare* message, then after a time interval, they again contact the coordinator and the default message is to abort.
- Since the coordinator does not wait to hear from the participants after deciding to abort a transaction, the names of participants need not be recorded in the abort log record.
- All abort log records can be appended to the log tail instead of doing a force-write.
- If a sub-transaction does no updates, it can respond to a prepare message by saying that it is a reader. Participants need not write a log record, since it does not modify the database.
- If a coordinator receives a reader message, it treats the message as *yes* but does not send any message.
- If all sub-transactions are readers, then the second phase is not required.

This refined 2PC is known as *two-phase commit with presumed abort*.

12.8.4 Three-Phase Commit (3PC)

3PC introduces a third phase in the 2PC protocol so that a transaction is not blocked during failures. The steps are as follows:

- The coordinator sends a *prepare* message to each participant.
- Participants respond with either yes or no.
- If the coordinator receives all yes then sends the message *pre-commit*, else *abort*.
- Each participant must *acknowledge* the commit or abort message.
- The coordinator on receiving an *acknowledge* message from maximum participants, sends a *commit* message.
- The coordinator writes the end record in the log file for the transaction.

In 3PC, there are three rounds of messages: (a) Voting round, (b) Pre-commit round, and (c) Termination round. It is assumed that there is no network partitioning and at any point, at least one site must be up.

Phase 1: Voting

- The coordinator sends out a *prepare* message and receives yes votes from all participants.

Phase 2: Pre-commit

- The coordinator sends a *precommit* or *abort* message to all participants. Most participants respond with *ack* message.

Phase 3: Termination

- When a sufficient number of messages have been received from the participants, the coordinator force-writes a *commit* log record and then sends a *commit* message to all.

In 3PC, the coordinator effectively postpones the decision till it is sure that a sufficient number of sites know about the decision to commit. If the coordinator fails subsequently, then these participants can communicate with each other and then find out whether the decision is to commit or abort. The decision to abort is taken if none has received a *pre-commit* message.

The first phase of 2PC and 3PC are the same, but the second phase of 2PC is divided into two phases: *pre-commit* and *termination*. Due to the pre-commit phase, the transaction is not blocked, even if the coordinator fails. The participants communicate with each other and then take the decision to commit/abort, as the case may be.

However, there are certain drawbacks. The 3PC imposes a significant additional cost of message communication during the normal execution and requires that the communication link failures do not lead to a network partition.

12.8.5 Network Partitioning

Though we assumed that there is no network partitioning, in reality, network partitions do occur, and that may affect the 2PC protocol. The following conditions may be considered:

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with the failure of the coordinator.
  - There are no negative effects, but the sites may still have to wait for a decision from the coordinator.
- The coordinator and the sites are in the same partition, as the coordinator thinks that the sites in the other partition have failed, and follow the usual commit protocol. Again, there are no negative effects.

12.9 Mobile Databases

The demands on mobile computing are increasing and there is a growing need to provide support to mobile workers. There are data requirements as if the workers are accessing data in the office but in reality they are working from remote locations such as homes, clients' premises, or even while travelling. The office accompanies the worker in the form of a PDA (Personal Digital Assistant), laptop, BlackBerry or any other Internet access device. With the rapid expansion of cellular, wireless and satellite communications, mobile users can now access any data from anywhere and at any time.

Due to practical limitations, security issues, and communication costs, it is not feasible to establish online connections for as long as the users want. Mobile databases offer a solution to these limitations.

With mobile databases, users have access to corporate data from their laptops, PDAs or any other Internet access device at remote sites. Figure 12-19 depicts a typical mobile database environment.

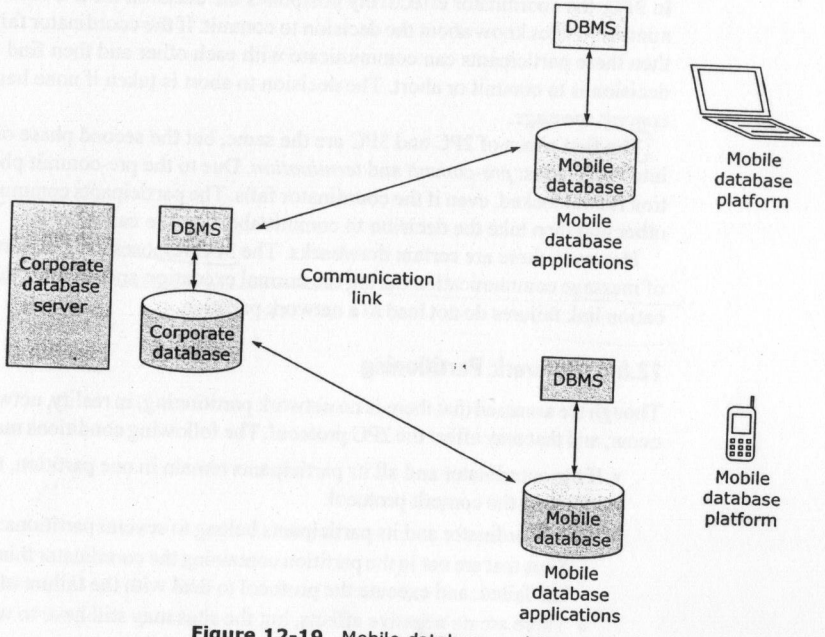


Figure 12-19 Mobile database architecture

The various components of a mobile data environment include:

- A corporate database server and DBMS that manages and stores the corporate data and provides corporate applications.

- A remote database and DBMS that manages and stores the mobile data and provides mobile applications.
- A mobile database platform that includes laptop, PDA, or any other Internet access device.
- Two-way communication links between the corporate and mobile DBMS.

Depending on the specific requirements of mobile applications, one of the following situations can occur:

- The user of a mobile device may log on to the corporate database server and work with the data.
- The user may download the data and work with it on a mobile device.
- The user may upload the data captured at the remote site to the corporate database.

The communication between the corporate and mobile databases is intermittent and established for short intervals of time. Two major issues associated with mobile databases is the management of the mobile database and the communication between the mobile and corporate databases.

12.9.1 Mobile Database Management Systems

Currently, all major DBMS vendors offer a mobile DBMS that is capable of communicating with a range of relational database services which require limited-computing resources to match those currently provided by mobile devices.

The additional functionality required for mobile DBMSs includes the ability to:

- Communicate with the centralized database server and mobile devices
- Replicate data on the centralized database server and mobile device
- Synchronize data on the centralized database server and mobile device
- Capture data from various sources like the Internet
- Manage data on the mobile device
- Analyze data on a mobile device
- Create customized mobile applicaitons

12.10 Case Study: Distribution and Replication in Oracle

Oracle is a commonly used DBMS to implement replication and distributed functionalities. Oracle does not support *fragmentation transparency*, but it supports *location transparency*.

12.10.1 Oracle's Distributed Functionality

The Oracle DBMS is divided into two parts: front-end as the *client* portion and back-end as the *server* portion. Oracle defines a relation with the help of *tables*, *rows*, and *columns*. We discuss the following:

- Connectivity
- Global database names
- Database links
- Referential integrity
- Heterogeneous distributed database
- Distributed query optimization

Connectivity Oracle supplies Net8 data access application to support client-server communication. It enables both client-server communication across any network supporting distributed processing and DDBMS capability. Net8 is required to establish connection between a process and a database instant executing on the same machine. Net8 also converts character sets or data representations if necessary, at the operating system level. Net8 sends the connection request to the Transparent Network Substrate (TNS). TNS chooses the server which should handle the request and sends the request further using the correct network protocol (TCP/IP or SPX/IPX). Net8 also handles the communication between heterogeneous databases.

The Oracle Names product stores database information in a distributed environment at a single location. When an application requests a server connection, the Oracle Names repository is consulted to determine the server location and then the connection is set up between the client and the server.

Global database names Each distributed database is given a unique name called the *global database name*, created by prefixing the network domain name with the local database name. The network domain name follows a standard Internet connection. For example, the property databases at Mumbai and Pune can be named as *RENTALS.MUMBAI.COM* and *RENTALS.PUNE.COM*

Database links The distributed databases in Oracle are connected together with database links. It makes the remote data at different databases available for queries and updates. A database link should be given the same name as the global database name. For example,

CREATE PUBLIC DATABASE LINK RENTALS.PUNE.NORTH.COM

This database link can be used to refer to tables and views on the remote database by appending *@database* link to table or views. Consider the following SQL statement:

*SELECT * FROM Emp@RENTALS.PUNE.NORTH.COM*

*UPDATE Emp @RENTALS.PUNE.NORTH.COM SET pay = pay*1.5*

A user can access tables owned by other users in the same database by preceding the database name with the schema name.

*SELECT * FROM Supervisor.Viewing @ RENTALS.PUNE.NORTH.COM*

This statement connects as the current user to the remote database and then queries the viewing in the Supervisor schema.

Referential integrity Oracle does not permit referential integrity constraints to be defined across databases in a distributed system. In simple words, it means that a declarative referential constraint on one table cannot specify a foreign key that references a primary key on a remote table.

Heterogeneous distributed databases In an Oracle distributed database system, at least one of the DBMSs is a non-Oracle system. Given an heterogeneous services software and a non-Oracle, system-specific heterogeneous services agent, Oracle can hide the distribution and heterogeneity from the user. The heterogeneous services agent communicates with the non-Oracle system and with the heterogeneous services component in the Oracle server. On behalf of the Oracle server, the agent executes the SQL procedure and the transactional requests at the non-Oracle system. Two such software tools are:

- *Transparent gateways:* These gateways provide SQL access to non-Oracle DBMSs like Informix, Sybase, DB2/400, DB2 for OS/390, SQL server, etc. These gateways run on the machines with a non-Oracle DBMS as opposed to an Oracle server.
- *Procedural gateways:* These gateways use remote procedure calls (RPCs) to applications built on non-Oracle DBMSs.

The features of heterogeneous services include:

- Distributed transactions
- Transparent SQL access
- Procedural access
- Data dictionary translations
- Pass-through SQL and stored procedures
- National language support

Distributed query optimization A distributed query is decomposed into multiple sub-queries by the Oracle DBMS and then sent to remote databases for execution. The remote databases execute the queries and then send the result back to the local node. The local node does further postprocessing and returns the result to the user. Only the necessary data is extracted from the remote tables so that lesser data is transferred and the communication cost is reduced. Distributed query optimizer uses Oracle's cost-based optimizer.

12.10.2 Oracle's Replication Functionality

Oracle supports both synchronous and asynchronous replication through *Oracle advanced replication.* It allows tables and supporting objects such as views, triggers, packages, indexes, and synonyms to be replicated. The standard edition of Oracle has a master site and multiple slave sites. The master site can replicate the changes to slave sites. In the enterprise edition of Oracle, there are multiple master sites and any one of them can be updated. Let us first discuss the types of replication that Oracle supports.

Oracle supports four types of replication:

- *Read-only snapshots:* A master table is copied on one or more remote databases. Changes in the master table are reflected in the snapshot tables whenever they are refreshed. It is sometimes known as *materialized views*.
- *Updatable snapshots:* It is similar to read-only snapshots except that the snapshot sites are able to modify the data, and then update the master site.
- *Multimaster replication:* A table is copied to one or more databases, where the table can be updated. The DBA for each replication group fixes an interval for periodically sending modifications to other databases in the group.
- *Procedural replication:* A call to the packaged procedure or function is replicated to one or more databases.

Replication groups These groups are created to organize the database schema objects required by a particular application. Replication group objects can come from many schemas and a schema can obtain objects from many replication groups, but an object can be a member of only one group.

Replication sites There can be two types of sites: *master site* and *snapshot site*. The *master site* maintains a complete copy of all objects in a replication group. All master sites in a multimedia replication environment communicate directly with one another to propagate the updates to data in a replication group. The *snapshot site* supports read-only and updatable snapshots of table data at an associated master site. Snapshots allow asynchronous distribution of changes to individual relations, collection of relations, views, or fragments of relations, as per a predefined schedule. Snapshots are updated by one or more master tables via individual batch updates, known as *refreshes*, from a single master site. A replication group at a snapshot site is called a *snapshot group*.

Refresh groups If two or more snapshot groups need to be refreshed simultaneously, Oracle allows refresh groups to be created. After refreshing all the snapshots in a research group, the data of all snapshots will correspond to the same transactional consistent point.

Refresh types Oracle refreshes the snapshots in the following ways:

- *Complete:* The managing server executes the snapshot's defining query. The result set of the query replaces the existing query.
- *Fast:* The managing server identifies the changes that occurred in the master table since the most recent refresh, and then applies them to the snapshot.
- *Force:* The managing server first executes a fast refresh, and then, if this cannot be done, performs a complete refresh.

Oracle also defines procedures for creating the snapshots. It keeps snapshot logs.

Updatable snapshots These snapshots are similar to read-only snapshots except that the snapshot sites are able to modify the data and send them back to the master site. The snapshot site determines the frequency of refreshes and the frequency with which the updates are sent back to the master site.

Multimaster replication It means a table is copied to one or more remote databases, where the table can be updated. Modifications are pushed to other databases at an interval determined by the DBA.

Summary

The idea behind developing a distributed database system is to integrate the operational data of an organization and provide controlled access to the data. A DDBMS system is actually a software system that manages distributed databases in a manner transparent to the user. It is split into a number of fragments, and each fragment is stored on one or more computers, sometimes separated geographically. Fragments may be replicated and kept on separate sites. Distributed processing, distributed DBMS, and parallel DBMS, are all distinct from each other. A parallel database is running across multiple processors and disks designed to execute operations in parallel. A distributed database uses the client-server architecture. A distributed processing system consists of a number of processing elements interconnected by a computer network that cooperate to access the data stored in a centralized database. There are different types of databases such as homogeneous DBMS, heterogeneous DBMS systems, open database systems, and multi-database systems.

A DDBMS hides the distributed nature from the user and provides an acceptable level of performance, but in that process, requires more complex procedures than a centralized database. Distributed DBMS architectures mainly follow one of the three approaches, viz. client-server systems, collaborating servers, or middleware systems.

The data in a distributed database is fragmented into vertical or horizontal parts and then stored at remote sites, so that message passing costs are reduced. Similarly, some fragments are replicated and stored at suitable sites. Fragmentation can be either horizontal or vertical.

Replication is another technique in which some relations or their fragments are replicated and stored in multiple sites. An entire relation can be replicated in one or more sites. Once the fragments are replicated, the system has to keep track of the replicated fragments through a distributed catalog. There are various approaches to catalog management: centralized global catalog, replicated global catalog, dispersed catalog and local master catalog. The naming system should also give unique names to each replicated or fragmented relation.

Whenever a distributed DBMS system is being used, users should be able to write queries irrespective of how the relation is fragmented or replicated. The DBMS should compute the relation as needed by locating suitable copies of fragments, joining vertical fragments if required, and taking union of horizontal fragments. This ensures distributed data independence. Distributed query processing is more complex than query handling in a centralized DBMS. A query may require data from more than one site. The data transfer costs for distributed query processing includes counting the number of page I/Os, and also counting the number of pages sent from one site to another because communication costs are a significant component of the overall cost in a DDBMS.