UNIT 4 DISTRIBUTED DATABASES

Structure **Page Nos.**

4.0 INTRODUCTION

Relational Database Management Systems are the backbone of most of the commercial database management systems. With the client-server architecture in place they can even be used in local and global networks. However, client-server systems are primarily centralised in nature–that is, the data is under the control of one single DBMS. What about applications that are distributed, where large number of transactions may be related to a local site, and where there are also a few global transactions? In such situations, it may be advisable to use a distributed database system. We have already introduced the concept of distributed databases in the MCS-023, Block 2, in this unit we enhance concepts that were introduced in that course.

In this unit, we will discuss the classification of distributed databases, its functions and architecture, distributed query design and the commit protocol. Towards the end of the unit we will also introduce the concept of replication servers.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- define the classification of distributed databases;
- identify the functions and architecture of distributed database management systems;
- state the principle of design of distributed database management system;
- create distributed query processing;
- explain concurrency management in distributed database systems;
- describe distributed commit protocols, and
- explain the concept of replication servers.

4.2 CENTRALISED Vs. NON-CENTRALISED DATABASES

If the data of an organisation is stored at a single location under the strict control of the DBMS, then, it is termed as a centralised database system. A centralised system has the advantage of controlling data under a central authority, so, the chances of integrity loss, inaccuracy are very low. In such a system it is easier to enforce business rules and the system can also be made very secure.

However, consider the situation when the organisation is geographically dispersed and the data is created and used at many distributed offices of the same organisation. In such a case the centralised database may act, as a bottleneck as all accesses to data has to be from the central site. The reliability and availability of such a system will not only be dependent on the centralised system but also on the communication links. The load on the central system may also be very high if multiple transactions are going on. Compounding the problem would be the delay that may be caused because of source access through a communication channel.

On the other hand, if we keep the data in databases at local ends that could to an extent solve these problems, as most of the data access now can be local, however, it adds overheads on part of maintaining databases at several sites. The sites must coordinate if a global query is to be handled. Thus, it will increase the overall complexity of the system.

Decision to maintain a centralised or non-centralised data will depend entirely on the nature and structure of an organisation.

4.3 DDBMS AND ITS FUNCTIONS

This is the era of distribution where the terms prefixed with “distributed”, “distributed database” and “distributed processing” are attracting a lot of attention. But how are these two different? Let us first define the two terms.

Distributed Database: A distributed database defines the distribution of a database application system in many different computer sites supporting their own DBMS environment. However, these databases from the viewpoint of a user are a single application.

Distributed Processing: Distributed processing is the distribution of various tasks of an application processing system among different computers on a network.

But how are they related to each other? The relationship can be explained by the fact that a database application may distribute front-end tasks such as display, formatting, etc. to the client site, whereas, the database sever becomes the backend. Many clients may share a backend server. Thus, distributing the tasks of an application that require a database system. Thus, a client-server system is typically a distributed system.

So, a distributed database system may employ a distributed processing architecture.

A distributed database system is managed by a distributed database management system (DDBMS). A DDBMS can be classified as a homogeneous or heterogeneous system. Let us explain these in more detail in this section.

4.3.1 Heterogeneous Distributed Databases

In a *heterogeneous* distributed database system various sites that are participating in the distributed database may be using different DBMSs, or different data model, or may not even be a database system (non-system). Thus, the major challenge for the heterogeneous distributed database system would be to make these systems appear as a single application system for the user.

A heterogeneous database system may have to allow data access from various type of data and thus, have to know about such data formats, file formats and management related details for such data. A heterogenous database system may need to develop some standard forms that could be used for bringing the heterogeneous sources of data to a common interface. Therefore, heterogeneous DDBMS are very complex in nature. A detailed discussion on this topic is beyond the scope of this unit. However, let us try to list some of the important issues that need to be addressed by the Heterogeneous Distributed Database Systems:

- *Support for distributed transaction handling:* A heterogeneous DDMS must deal with transactions that are required to be executed on various DBMSs. The main concern here is to implement the commit protocol across the DBMS to ensure proper completion/termination of a transaction.
- *SQL access through DBMSs and Non-DBMSs:* Heterogeneous DDBMS must integrate data that may be available in non-database systems along with the data that is available in the DBMS environment. The key here is that any SQL query should generate the desired result even if, data is available in the non-database system. This action is to be supported transparently, such that the users need not know the details.
- *Common interface for Procedural Access:* A heterogeneous DDMS must support a common interface for any procedural access like messaging and queuing systems. One such implementation interface may be the use of Host language-SQL remote procedure calls.
- *Support for Data Dictionary translation:* One of the key support needed for DDBMS is to homogenise the data dictionaries of the systems as well as the non-database systems. One of the basic requirements here would be to translate a reference to a data dictionary or, a DBMS into a reference and then, to a non-system's data dictionary.
- Heterogeneous DDBMS should allow accessing data of non-systems directly without the use of SQL; on the other hand it should also allow access to stored procedures using remote procedure calls of the host languages.
- It should have support for the various different kinds of character sets that are used across non-systems and DBMS.
- It should support the use of distributed Multi-Threaded Agents. The Multi-threaded agents help in reducing the number of required server processes. It should also provide a graphic interface to access those agents.

4.3.2 Homogeneous Distributed Databases

The homogeneous distributed database systems contain identical DBMS across cooperating sites. These sites are aware of each other and need to compromise a little on their individual autonomy. In this unit, we have discussed details regarding these types of distributed database systems.

4.3.3 Functions of a DDBMS

A DDBMS primarily is built on a centralised relational database system, thus, it has the basic functionality of a centralised DBMS along with several other functionalities such as:

- It allows access to remote sites using standard SQL interfaces, thus allowing transfer of queries or components of queries as well as data of the database over a network.
- It has complete control over the system catalogue that stores the details of data distribution. Please note that this catalogue itself may however, be distributed.
- It performs distributed query processing, optimisation and evaluation. Please note that distributed query processing involves even access to remote data over the network.

- One of the major functions of the distributed database system is the control of distributed concurrent transactions to maintain consistency of the data that may be replicated accross various sites.
- Access to the databases under the replicated environment should be controlled. Thus, a distributed database management system should follow proper authorisation/access privileges to the distributed data.
- Distributed recovery mechanisms in addition to dealing with the failure of participating sites may also need to deal with the failure of communication links.

Before discussing some specific details of the DBMS, let us first look at the reference architecture of DDBMS.

4.4 REFERENCE ARCHITECTURE OF DDBMS

Distributed database systems are very diverse in nature; hence, it may be a good idea to define reference architecture for such database systems. For the DDBMS reference architecture may be looked at as, an extension of ANSI SPARC three level architecture that we have discussed in Unit 1 of MCS 023. The ANSI SPARC architecture has been defined for the centralised system and can be extended for the distributed database systems with various levels defined for data distribution as well.

Reference architecture may also define the following additional schema for distributed database systems:

- A set of global schemas that would define application interface. Thus, a global schema can be further divided into:
  - Global external schema, and
  - Global conceptual schema.
- Fragmentation and allocation schema will determine various fragments and their allocation to various distributed database sites.
- Finally, it would require basic schemas as proposed by ANSI/SPARC architecture for local databases. However, some modification would be. Let us, explain the various schemas of reference architecture in more detail.

Global External Schema

The basic objective of providing a global external schema is to define the external application interface for a distributed database application. This level is similar to the external schema as defined for the ANSI SPARC architecture. Thus, this interface must support the concept of data independence. However, in this case, it is logical data independence that is provided to a database application from the distributed data environment.

Global Conceptual Schema

A Global conceptual schema defines logical database application as if there is, no distribution of data. Conceptually, this schema is almost similar to, the conceptual level of ANSI/SPARC architecture. Thus, this schema defines the entities, relationships, and constraints including security and integrity constraints for global database application. This level is responsible for ensuring physical data independence between an application and the data distribution environment where it is to be implemented.

Fragmentation and Allocation Schema

Fragmentation and allocation schema is required to describe the data fragments that have been created for the distributed database, alongwith, their replicas that have been allocated to various database sites. This schema, thus, basically is for the identification of data distribution among the distributed sites.

Local schemas

The local schema at each site can consists of conceptual and internal schemas. However, to map the fragmentation and allocation schema to local external objects may require local mapping schema. It is this schema (that is local mapping schema), that hides the details of various DBMSs, thus, providing the DBMS independence. This schema mapping, however, would be minimal for a homogeneous distributed database systems. *Figure 1* shows the reference architecture of DDBMS.

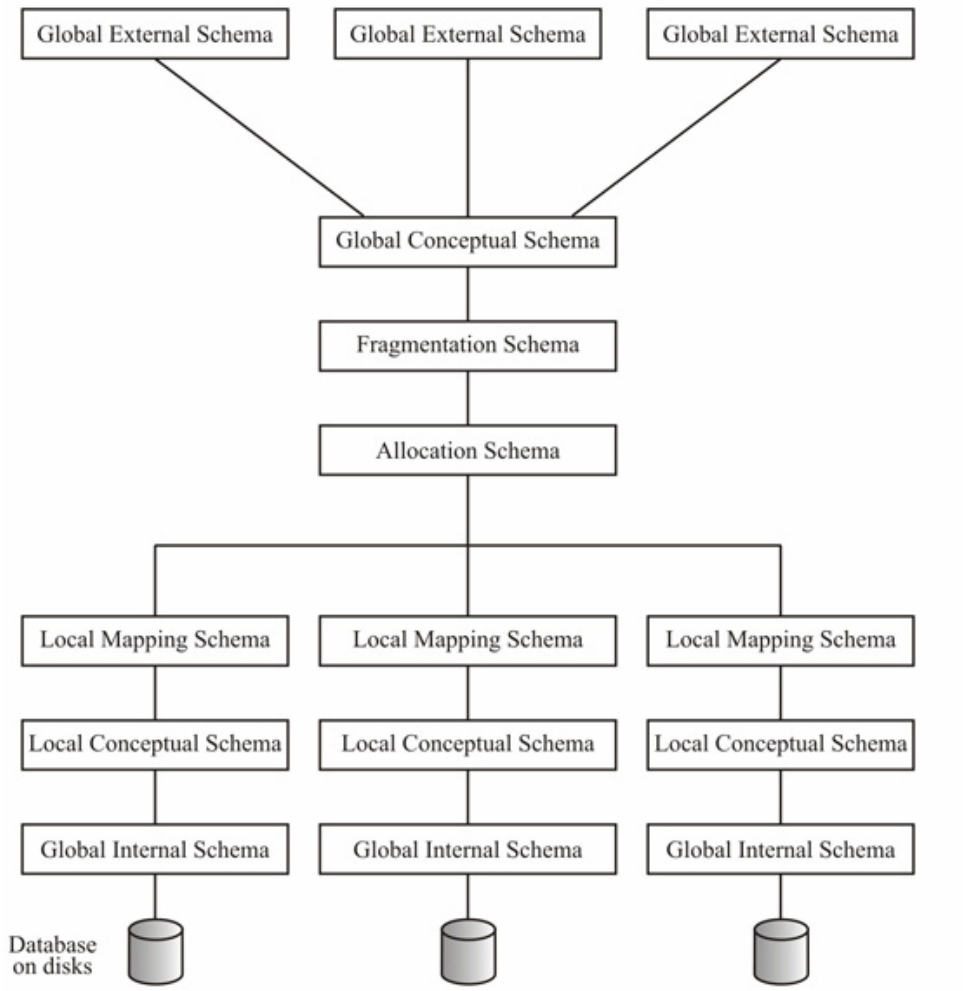


Figure 1: Reference architecture for DDBMS

Component Architecture of DDBMS

From the viewpoint of commercial implementation a DDBMS would require client server implementation with the backbone of a network. Thus, such an architecture would support:

Local DBMS Component (Clients and Servers): At a local database site a DDBMS communicates with a local standard DBMS. This site therefore can have its own client server architecture. A computer on such a site (also referred to as a node) may be a client or server depending on its role at that movement of time. For example, in *Figure 2* (this figures is same as that of *Figure 2* unit 4, Block-2 of MCS-023) the computer that has the HQ database will act as a database server if, we request data for local sales, however, it will act as a client if we request data from the Employee table which is not available on this site.

Data Communications Component: Although we have the desired client and server components, the question that now arises is related to communication. How will the

client and the server component communicate with each other? This communication should be established with remote clients may not be connected to a server directly. It is the data communication component that enables direct or indirect communication among various sites. For example, in the *Figure 2* the transactions that are shown occurring on Site 1 will have direct connection to Site 1, but they will have an indirect communication to Site 2, to access the local sales data of that site. Thus, a network is necessary for a distributed database system.

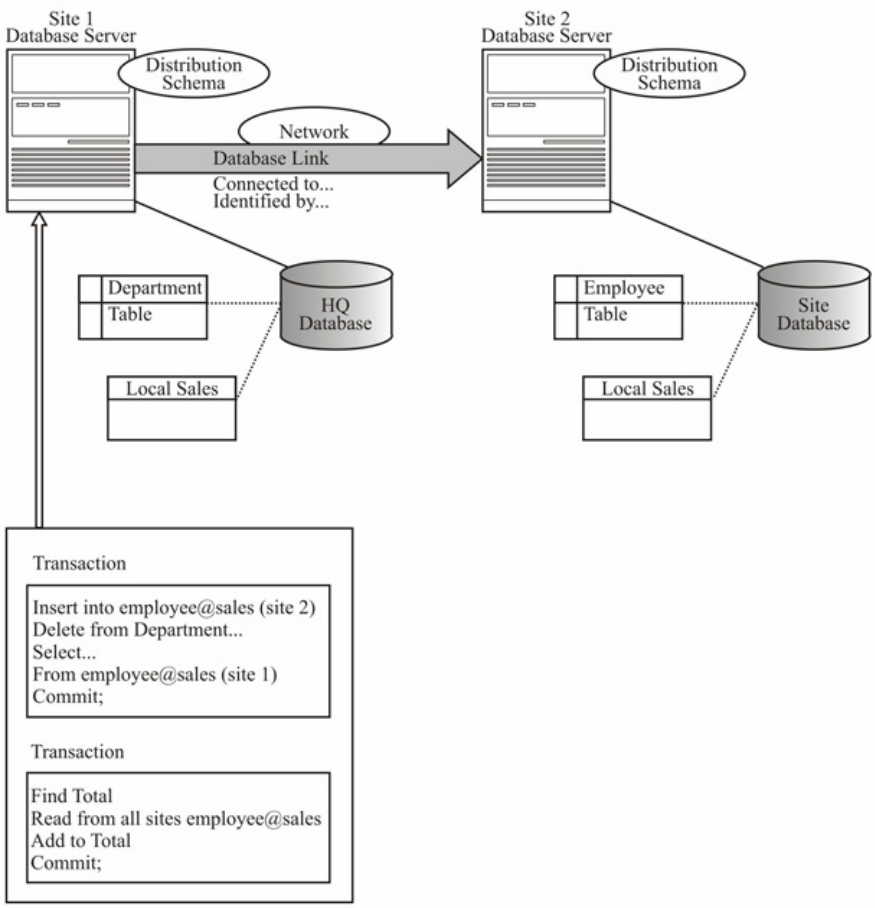


Figure 2: Component architecture of DDBMS

Thus, we have clients, servers and the network in the distributed databases. Now the question is how to use names in this network of a DDBMS sites? The names on a DDBMS may be provided using a global directory service. To allow easy access to the configuration of data, a networks name server can be used to provide information on the repositories of the databases and their data distribution.

Global System Catalogue: One of the ways of keeping information about data distribution is through the system catalogue of the DBMSs. In a DDBMS we need to keep a global system catalogue. This catalogue itself might be distributed in implementation. In addition to basic information about the databases it should keep information on data fragmentation and replication schema.

Database Links: A database in a distributed database is distinct from the other databases in the system. Each of these distributed databases may have their own global database name. All these database components would also have certain local names. Now, the questions are: How reachout from one database in a distributed database to another database and how does one resolve the names of various schema elements in the distributed databases that are available across the network. *Figure 3* represents a hierarchical arrangement of databases components that are available on a network. It shows the hierarchical arrangement of a hypothetical distributed database of some Universities.

As far as the first question is concerned, an application can travel from one database system to another by using database links, which is an unidirectional path from one database to another. Such links must be transparent from a user point of view.

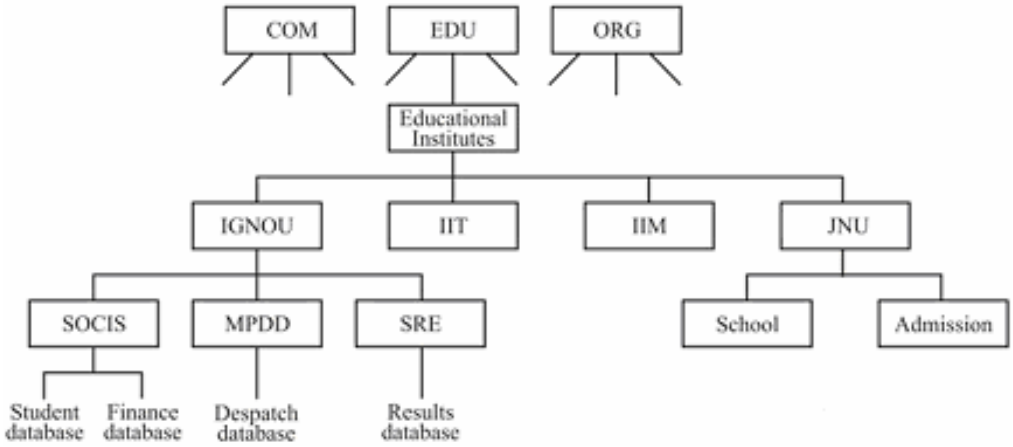


Figure 3: Network directories and global database naming

Now, as far as naming issues are concerned, the names of the database may be resolved by specifying the path. For example, in the *Figure 3*, the name for the result database in IGNOU – SRE may be RESULT may also be the same for the JNU, which database for storing the result of the student. However, these databases will be referred to as:

RESULT.SRE.IGNOU.EDU_INST.EDU

(Please note that EDU_INST represents "Educational Institutes" block in *Figure 3*)

and

RESULT.SCHOOL.JNU.EDU_INST.EDU

☞ Check Your Progress 1

1) What additional functions does a Distributed Database System have over a centralised database system?

……

……

……

……

2) How does the architecture of DDBMS differ from that of a centralised DBMS?

……

……

……

……

4.5 DISTRIBUTED DATABASE DESIGN AND DISTRIBUTED QUERY PROCESSING

Distributed database design has already been covered in MCS-023. Let us recapitulate some of the salient features of the distributed design.

The distributed design is primarily based on the nature of local and global transactions. Some of the key decisions taken for distributed database design, in addition to a centralised database design are:

- *data fragmentation* is concerned with the fragmentation of global data into component relations, and
- *data replication* determining the manner in which fragmented data is to be duplicated at several sites such that it supports ideal performance for data updating and query processing. It also ensures database availability and reliability.

Replication is a very important concept, so let us now talk about data replication in detail, next.

Data Replication

A distributed database without replication will have exactly one copy of a data item stored on some site. However, such an implementation would not offer advantages of a distributed database, (that are reliability and availability). A replicated distributed database will result in multiple copies of the data that will be stored on different sites. This replicated data will cause an increase in the overheads of data management, but will improve the availability of the system. For, even if one site with the data is down for some reason, the data will still be available from another site. In addition, data replication improves the performance of the distributed system. How? Well it is primarily because data is replicated with some logic, the chances are that a distributed application may find the remote data replicated on the local site, and thus, need not access a remote server. This in turn, may reduce network traffic and improve performance.

Distributed Query Optimisation

The concept of query optimisation in a distributed database system is the enhanced form of query optimisation that exists in the centralised databases. One additional but major objective of the distributed query optimisation is to minimise data transfer among the sites for distributed queries.

To fulfil this objective a distributed query optimiser would require some cost based optimisation that transforms SQL queries such that only the necessary data from a remote site is transferred over the network. It will also require the processing of the data at remote sites. The purpose of this processing would be that instead of a complete table only the data that is required for the query, will be transferred. Thus, reducing the amount of data transfer. Let us explain this with the help of an example:

Consider the following situation where two sites contain information as:

Employee table at site 1

empno	emp-name	Phone	Job
1001	Sanjay	921000001	Salesperson
1002	Vijay	922000001	Salesperson
1003	Rahul	931000001	Office
1004	Ajay	…..	Office
1005	Kamal	…..	Clerk
…	…….	….	…….
…..	…….	….	
10001	Anurag		Research

Sales table at site 2

empno	item-code	quantitysold
1001	A	5000
1002	B	7000
1001	B	6000
1009	A	7000
1010	D	5000

Now consider a query that is submitted at **site 2.** Find the details of the employee who have sold more than 6000 quantity of item B. The SQL query to evaluate it could be:

```
SELECT e.empno, e.emp-name, e.phone, e.job
FROM employee@site1 e, sales@site2  s
WHERE e.empno=s.empno AND s.item-code=B AND quantitysold>6000;
```

Now, to execute this query by simply transferring employee table from site 1 to site 2 may be very expensive, likewise transfer of sales table to site 1, taking the join at site 1 and sending the result back to site 2 may also be very expensive. A possible optimal solution for the query may be:

(i) Find those tuples at site 2 which satisfy the sales conditions at site 2; now project this table on empno.

```
SELECT empno
FROM sales@site2
WHERE item-code=B AND quantitysold >6000
```

This will result in a very small table. Let us store this result in the TEMP table.

(ii) Now transfer this TEMP table to site 1 and take a join there as:

```
SELECT *
FROM employee@site1 e, TEMP
WHERE e.empno = TEMP.empno
```

(iii) Now send the result of step (ii) back to site 2

Thus, this method reduces the cost of communication and hence may enhance performance.

Similarly, an update statement that may effect data at many sites may be partitioned and sent accordingly to different site requirements.

A distributed database system supports may queries, updates, procedure calls and transactions. Let us define them briefly.

A *remote query* accesses information from one remote site. It may access one or more tables at that site. For example, the following query submitted at site 2:

```
SELECT * FROM employee@site1;
```

A *remote update* modifies data at one remote site. It may modify one or more tables at that site. For example the following query issued at site 2:

```
UPDATE employee@site1
SET phone= '29250000'
WHERE empno=10001;
```

A *distributed query* retrieves information from more than one site. Query on sale of an item, as given, while discussing query optimisation is an example of a distributed query.

A *distributed update* modifies data on more than one site. A distributed update program may be written by using an embedded SQL in a host language. You can also write a procedure or trigger. A distributed update would access and update data on more than one site. In such updates program statements are sent to remote sites and execution of a program is treated as a single outcome viz., success or failure.

The *Remote Procedure Calls (RPCs)* is a procedure written in the host language using embedded SQL and are used to perform some tasks at the remote database server site. This call is invoked when a program calls a remote procedure. The local server passes the parameters to such remote procedures.

Remote Transaction contains one or more statements that references/modifies data at one remote site.

A *distributed transaction* includes one or more statements that references/ modifies data on two or more distinct sites of a distributed database.

Having discussed queries and the basic definitions of different types of queries and transactions on a distributed database, we are now ready to discuss concurrency control mechanism used for distributed databases.

4.6 DISTRIBUTED CONCURRENCY CONTROL

The basic objectives of concurrency control mechanism in a distributed database are:

(1) to ensure that various data items and their replications are in a consistent state, when concurrent transactions are going on at the same time, and
(2) to ensure that all the concurrent transactions get completed in finite time.

The objectives as above should be fulfilled by the concurrency control mechanism under the following:

- failure of a site or communication link,
- strict performance guidelines supporting parallelism of queries, and minimal storage and computational overheads, and
- dealing with the communication delays of the networked environment.

Despite the above considerations it should support atomicity.

A distributed database system has to deal with the following two types of problems that occur due to distributed concurrent transactions:

- Concurrent transactions related problems that occur in a centralised database, (namely, lost updates, read uncommitted and inconsistent analysis).
- In addition, concurrency related problems arise due to data distribution and replication. For example, the need to update a data item that is located at multiple locations. Such data item require to be updated at all the locations or on none at all.

Let us discuss concurrency related issues in distributed databases in more details.

4.6.1 Distributed Serialisability

Distributed serialisability is the extension of the concept of serialisability in a centralised database system under data distribution and replication. It is defined as follows:

> If the transaction schedule at each site is serialisable, then the **global schedule** (the union of all such local schedules at various sites) will also be serialisable if and only if the order of execution of transactions in the global schedules does not violate the order of execution of transactions in any of the local schedules.

Like centralised databases, concurrency control in the distributed environment is based on two main approaches−locking and timestamping. We have discussed these two techniques in unit 2 of this block, but just to recollect −locking ensures that the concurrent execution of a set of transactions is equivalent to *some* serial execution of those transactions. The timestamp mechanism ensures that the concurrent execution of a set of transactions is in the same order of the timestamps.

In case, a distributed database has only data fragmentation and no replication, then for remote transactions, we can directly use locking or timestamping protocols that are used for a centralised database, with modifications for handling remote data. However, to deal with distributed transactions and data replications we need to extend these protocols. In addition, for the locking protocol we need to make sure that no deadlock occurs as transactions are processed on more than one site. Let us discuss these mechanisms in more detail.

4.6.2 Locking Protocols

The basic locking protocol that is used in the centralised system – the two-phase locking (2PL) has been extended in the distributed system. Let us look at some of the extensions of the two-phase locking protocol.

Centralised 2PL

As the name of this protocol suggests that in this protocol a single site known as the central site maintains the entire locking information. Therefore, it has only one lock manager that grants and releases locks. There is only one *lock manager,* for the entire distributed DBMS that can grant and release locks. The following are the sequence of operations in the centralised 2PL protocol of a global transaction initiated at site S_i:

(1) A transaction starts at a site S_i. This start site is mostly assigned the responsibility of coordinating that transaction. Therefore, it is also known as the transaction coordinator.

(2) A transaction coordinator takes the help of the global system catalogue to find details of the data items needed by the transaction it is coordinating. It then divides the transaction into a number of sub-transactions.

(3) The coordinator makes sure that a transaction is committed as per proper commit protocols as discussed in section 4.7. In addition, an updating transaction needs to obtain locks on the data items it is updating during the life cycle of the transaction. The transaction needs to make sure that all copies of data items are being updated by it, are updated consistently. Thus, the transaction requests an exclusive lock on those data items. However, a transaction that requires only Reading of a data item, can read it from any copy, preferably the coordinator performs read from its own local copy, if the data item already exists.

(4) The sub transactions that are being performed at local sites have their own local transaction manager. The local transaction manager is responsible for making the request for obtaining or releasing a lock (following the rules of 2PL) to the centralised lock manager.

(5) The centralised lock manager grants the locks or asks local transactions to wait by putting them in the wait queue, after checking the status of the item on which lock request is made. In case the lock is granted then this acknowledgement is communicated to the local lock transaction manager.

This scheme can be modified by making the transaction manager responsible for making all the lock requests rather than the sub transaction's local transaction manager. Thus, the centralised lock manager needs to talk to only the transaction coordinator.

One of the major advantage of the centralised 2PL is that this protocol can detect deadlocks very easily as there is only one lock manager. However, this lock manager often, becomes the bottleneck in the distributed database and may make a system less reliable. This is, due to the fact, that in such a case, for the system to function correctly it needs to depend on a central site meant for lock management.

Primary Copy 2PL

In this scheme instead of relying on one central site the role of the lock manager is distributed over a number of sites. These lock managers are made responsible for managing the locks on the data items. This responsibility fixing is done by naming one of the copies of the replicated data as the **primary copy.** The other copies are called **slave copies**. Please note that it is not necessary that the site that is made responsible for managing locks on the primary copy is the same site.

In a scheme, a transaction coordinator requests the lock from the site that holds the primary copy so that the request can be sent to the appropriate lock manager. A lock

in this scheme may be granted only on the primary copy as this is the copy that will ensure that changes are propagated to the appropriate slave copies consistently.

This approach may be a good approach for distributed databases in which we do not have frequent updates, and, inconsistent values for some time are acceptable, as this scheme does not always result in consistent values of data in the primary and slave copies. The major disadvantage of this mechanism is that it complicates the process of deadlock detection, as there is many lock managers now.

This locking protocol has lower communication costs and better performance than centralised 2PL as it requires less number of remote locks.

Distributed 2PL

The basic objective of this protocol is to overcome the problem of the central site controlling the locking. It distributes the lock managers to every site such that each the lock manager is responsible for managing the locks on the data items of that site. In case there is no replication of data, this protocol, thus, will be identical to the primary copy locking.

Distributed 2PL is a Read-One-Write-All type of protocol. But what does this read one and write all protocol mean. It means that for reading a data item any copy of the replicated data may be read (as all values must be consistent), however, for an update all the replicated copies must be exclusively locked. This scheme manages the locks in a decentralised manner. The disadvantage of this protocol is that deadlock handling is very complex, as it needs to be determined by information from many distributed lock managers. Another disadvantage of this protocol is the high communication cost while updating the data, as this protocol would require locks on all the replicated copies and updating all of them consistently.

Majority locking

The distributed locking protocol is a natural protocol for the distributed system, but can we avoid the overheads on updates in such a protocol? Well, we need not lock all the replicated copies for updates; it may be alright, if the majority of replicas are locked. The lock manager still needs to be maintained at each site, but now, there would be less number of lock requests for update. However, now, the lock request would be needed for reading as well. On a read or write request from a transaction for a data item the following will be checked:

> Assume the request is for a data item having n replications, then read or exclusive locks are to be obtained for at least ($\lfloor n/2 \rfloor + 1$) replicas. Thus, for each data item more than half of the locks need to be obtained on the replicated data item instances by sending requests to that many lock managers. A transaction needs to obtain all the required locks in a time period, or else it may roll back the lock request. Please note, in this mechanism, a shared lock may be placed on a replicated data item instance by many transactions, but only one exclusive lock can be held on it. Also, the item, cannot be locked exclusively if at least one transaction holds a shared lock. Thus, a data item with all its replication may be locked in read or shared mode by more than one transaction, but only by one transaction in the exclusive mode.

The majority of locking protocols is complex to implement. It may not even be very useful for applications that require more reads and less writes as in this case, pf reading this protocol requires majority locks. Another important thing here is to consider that the number of successful lock requests is equal to the number of unlocking requests too. Thus, reading in this protocol is more expensive than in distributed locking.

4.6.3 Timestamp Protocols

Timestamping protocols are basically designed to model the order of execution of transactions in the order of the timestamps. An older transaction has a lower

timestamp and should be given the resources in case of a resource conflict. In a centralised system the timestamp can be generated by a single source. It is therefore, unique. But, how do you generate unique timestamps for a distributed database system? One possible solution here may be to generate unique local and global timestamps for the transactions. This can be done by producing a unique local timestamp and appending it with the information such as a unique site identifier. But where should we put this unique site identifier on the timestamp? Obviously, since the protocol deals with the timestamp order, the site identifier should be at the least significant bits. This would require synchronisation of local timestamps that are being produced at local sites. This synchronisation would require that messages pass between/among the different sites sharing its latest timestamp.

Let us explain the timestamp generation with the help of an example:

Consider two sites of a distributed database management system, producing the timestamps as integer numbers. Also assume that the sites have their unique site identifiers as 1 and 2 respectively. Assume that the site 1 have many transactions, whereas site 2 has a only few transactions. Then some of the timestamps generated by them would be shown in the following *Table*:

Site 1			**Site 2**		
Local timestamp	Site identifier	Unique Global Timestamp	Local timestamp	Site identifier	Unique Global Timestamp
1 2 3 4 5 6	1 1 1 1 1 1	1,1 2,1 3,1 4,1 5,1 6,1	1 2	2 2	1,2 2,2
MESSAGE FOR SYNCRONISATION SYNCHRONISES BOTH THE SITES TO HAVE NEXT LOCAL TIMESTAMP AS 7					
7 8 ..	1 1	7,1 8,1	7	2	7,2

4.6.4 Distributed Deadlock Management

A deadlock is a situation when some of the transactions are waiting for each other to free up some data items such that none of them is able to continue. The deadlock in a distributed database base management system is even more difficult to detect than that of a centralised database system. Let us demonstrate this with the help of an example.

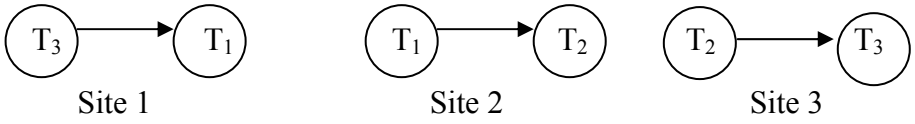
Consider three transactions T_1, T_2 and T_3 with the following locking requests:

Transaction T_1 Transaction Coordinator: Site 1 (S1) Sub transaction Site: S2	Transaction T_2 Transaction Coordinator: S2 Sub transaction Site: S3	Transaction T_3 Transaction Coordinator: S3 Sub transaction Site: S1
(S1) SLOCK (X) (S1) SLOCK (Y) (S2) XLOCK (SUM) (S2) SLOCK (Z) Send X, Y values to S2 (S2) SUM = X + Y + Z (S1) UNLOCK (X) (S1) UNLOCK (Y) (S2) UNLOCK (Z) (S2) UNLOCK (SUM)	(S2) XLOCK (Z) (S3) XLOCK (A) (S2) Z = Z - 1000 (S3) A = A +1000 (S2) UNLOCK (Z) (S3) UNLOCK (A)	(S3) XLOCK (A) (S1) XLOCK (X) (S1) X = X - 1000 (S3) A = A +1000 (S1) UNLOCK (X) (S3) UNLOCK (A)

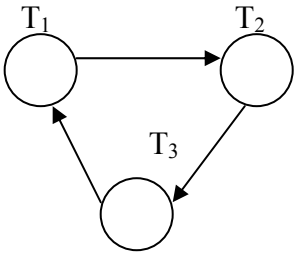
Assume that the three transactions start at the same time, then the lock assignments will be like:

At site S1: T_1 has shared lock on X and Y. The sub-transaction of T_3 is waiting for lock on item X that is held by T_1.
At site S2: T_2 has exclusive lock on Z, while the sub-transaction of T_1 is waiting for lock on item Z that is held by T_2.
At site S3: T_3 has shared lock on A. The sub-transaction of T_2 is waiting for lock on item A that is held by T_3.

Thus, the local wait-for graphs for the three sites are given in *Figure 4* (a). Please note that there is no cycle in the local wait-for graph at each site. However, if we combine the local wait-for graphs to form a global wait-for graph (please refer to *Figure 4* (b)) we detect a cycle of the form: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$ which indicates the deadlock situation.



(a) Local wait for graph



(b) Global wait for graph: Detects a deadlock

Figure 4: Distributed deadlock detection

Thus, to detect a deadlock situation in a DDBMS, we must make both local and Global wait-for graphs. Let us now discuss the three methods of detecting deadlocks in DDBMSs – Centralised, Hierarchical and Distributed deadlock detection mechanisms.

Centralised Deadlock Detection: The following are the steps for the centralised deadlock detection scheme:

- Appoint a single central site as the Deadlock Detection Coordinator (DDC), which constructs and maintains the global WFG.
- Each local lock manager must transmit its local WFG to the DDC.
- The DDC builds the global WFG and checks for cycles in this WFG.
- In case one or more cycles break each cycle by rolling back a transaction this operation will break the entire cycle.
- Restart this rolled back transaction and inform all the sub-transactions involved.

The information flow can be minimised in this approach by transferring the portions of local WFG that have been changed since the last transmission.
The centralised approach is a compromise on reliability in a distributed database system.

Hierarchical deadlock detection: This is a scheme based on the tree type architecture. In this approach the Global wait-for graph is constructed by passing local WFGs to the site above in the hierarchy. *Figure 5* shows a hierarchy of eight sites, S_1 to S_8.

Every level in the hierarchy is a combination of the previous level combined wait-for graphs.

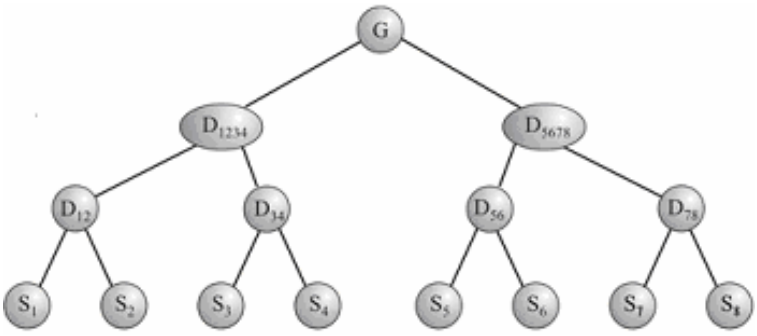


Figure 5: Hierarchical deadlock detection

The hierarchical approach is a complex approach but it reduces the dependence on a centralised detection site. It also reduces the local WFGs communication costs.

Distributed Deadlock Detection: There are many distributed deadlock detection algorithms. We will briefly explain one of these algorithms.

The distributed deadlock detection algorithm leaves the responsibility of deadlock detection to a local node where transactions are taking place. Each node has its local wait for graph, however, the information about potential deadlock cycles are passed by all other participating sites. Thus, they are able to detect possible deadlock situation. The basic algorithm for it is as follows:

(1) create the local wait for graph,

(2) add possible edges obtained from other sites that may cause deadlock,

(3) the local wait for graph now also contains locks on remote objects and the sub-transactions holding those locks, and

(4) determine the cycles, if it is to be found. There is a deadlock.

☞ Check Your Progress 2

1) What are the issues of query processing in a distributed system?

……

……

……

……

2) What are the serialisability requirements for a distributed database?

……

……

……

……

3) Why can we not detect deadlocks in a distributed database with a simple wait for graph?

……

……

……

……

4. 7 DISTRIBUTED COMMIT PROTOCOLS: 2 PC AND 3 PC

A transaction is a unit of work as well as the unit of recovery. A transaction has four basic properties – atomicity, consistency, isolation and durability. Thus, it is the responsibility of any DBMS to ensure that the transaction is executed as a single unit, that is, completely or not at all. It should leave the database in a consistent state, the effect of execution of one transaction should not be visible to other transactions till commit, and the transaction once committed cannot be undone. A DDBMS is also expected to preserve these properties of the transaction. However, in order to do so, it needs to do much more than the centralised databases. The key here is to design a commit protocol for distributed transactions. Let us analyse the commit protocol of distributed DDBMS in more details.

4.7.1 Two-Phase Commit (2PC)

In a DDBMS since a transaction is being executed at more than one site, the commit protocol may be divided into two phases. The basic objective of this *two-phase commit* protocol is to make sure that a transaction and all its sub-transactions commit together or do not commit at all. Please note that, a two-phase commit protocol ensures integrity and handles remote procedure calls and triggers.

A 2PC consists of two phases:

- voting phase and
- decision phase.

For each transaction, there is a coordinator who controls the 2PC. The sub-transaction sites act as participants. The logical steps of 2PC are:

- coordinator asks its participants whether they are prepared to commit the transaction, or not
- participants may votes to commit, abort or fail to respond within a time-out period,
- the coordinator makes a decision to commit if all participants say commit or abort even if one participant says so or, even if, one of the participant does not respond,
- global decision is communicated to all the participants,
- if a participant votes to abort, then it is free to abort the transaction immediately as the result of this voting will definitely be abort. This is known as a **unilateral** abort,
- if a participant votes to commit, then it waits for the coordinator to send the *global commit* or *global abort* message,
- each participant maintains its local log so that it can commit or rollback the local sub-transaction reliably. In 2PC participants wait for messages from other sites, thus, unnecessarily blocking the participating processes, even though a system of timeouts is used.

The steps for the coordinator in 2PC commit is as follows:

Coordinator

Phase 1

- Writes a *start-commit* record to its log file.

- Sends a PREPARE message to all the participants.
- Waits for participants to respond within a time period.

Phase 2

- If even a single participant returns ABORT vote, then write *abort* record to its log and send a GLOBAL-ABORT message to all the participants. It then waits for participants to acknowledge the global message within a time period.
- If a participant returns a COMMIT vote, then update the list of responses. Check if all participants have voted COMMIT, if so it writes a *commit* record to its log file. It then sends a GLOBAL-COMMIT message to all participants and then waits for acknowledgements in a specific time period.
- If all acknowledgements are received then the coordinator writes an *end-transaction* message to its log file. If a site does not acknowledge then the coordinator re-sends the global decision until an acknowledgement is received.

In Phase1 the coordinator should wait until it has received the vote from all participants. If a site fails to vote, then the coordinator assumes an ABORT vote by default and takes a decision to GLOBAL-ABORT and sends this message to all participants.

The steps at the participant in 2PL is as follows:

Participant

Phase 1

- On receiving a PREPARE message, a participant:
  - (a) writes a *ready to commit* record to its local log file and send a COMMIT vote message to the coordinator, or
  - (b) writes an *abort* record to its log file and send an ABORT message to the coordinator. It can now abort the transaction unilaterally.
- It then waits for the coordinator to respond within a time period.

Phase 2

- If the participant receives a GLOBAL-ABORT message, it writes an *abort* record to its log file. It then aborts the transaction and on completion of this step, it sends an acknowledgement to the coordinator. Please note in case this transaction has sent an abort message in phase 1 then it may have completed the unilateral abort. In such a case only the acknowledgement needs to be sent.
- If the participant receives a GLOBAL-COMMIT message, it writes a *commit* record to its log file. It then commits the transaction, releases all the locks it holds, and sends an acknowledgement to the coordinator.

What happens when a participant fails to receive the GLOBAL – COMMIT or GLOBAL – ABORT from the coordinator or the coordinator fails to receive a response from a participant? In such a situation a termination protocol may be invoked. A termination protocol is followed by only the active sites. The failed sites need to follow those when they recover. In practice, these protocols are quite complex in nature but have been implemented in most of the distributed database management system.

Figure 6 depicts the communication between the participant and coordinator.

Figure 7 shows the state of transitions during 2PC.

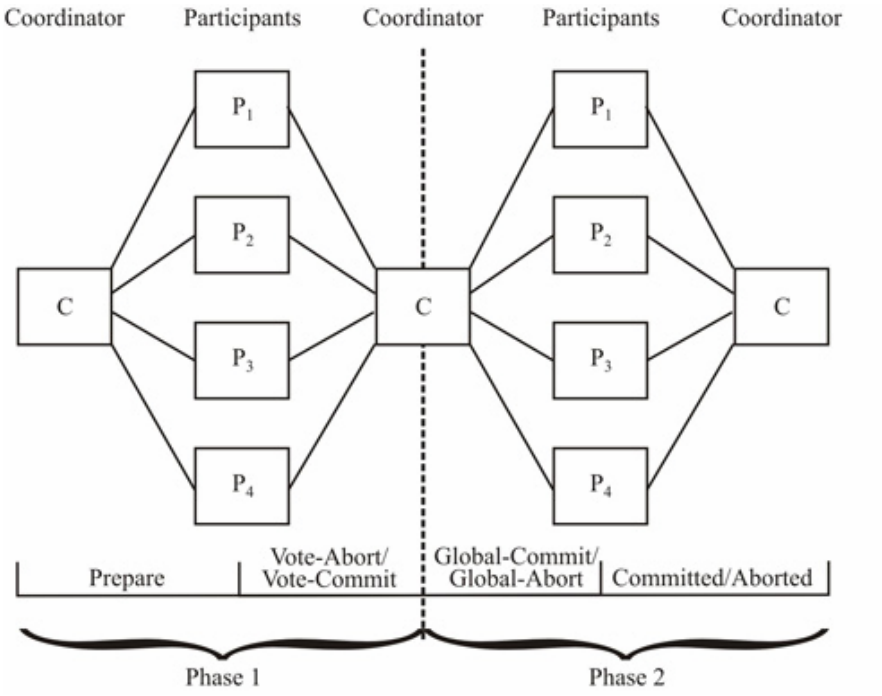


Figure 6: 2 PC Communication structure

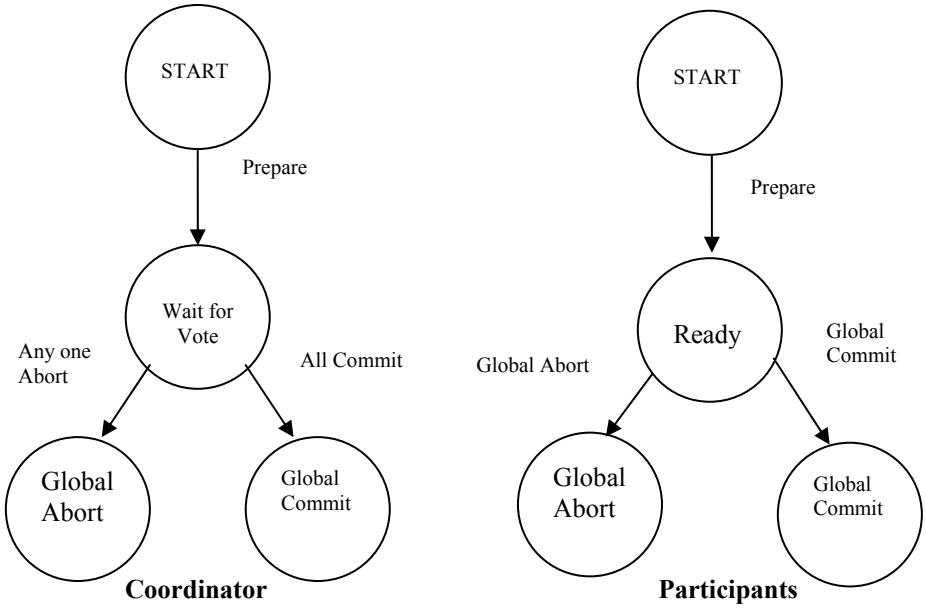


Figure 7: State Transition diagram in 2PC

4.7.2 Three-Phase Commit (3PC)

The 2PC is a good protocol to be implemented in a DDBMS. However, it has a small disadvantage−it can block participants in certain circumstances. For example, processes that encounter a timeout after voting COMMIT, but before receiving the global commit or abort message from the coordinator, are waiting for the message and doing nothing or in other words are *blocked*. Practically, the probability of blocking is very low for most existing 2PC implementations. Yet an alternative non-blocking protocol, – the **three-phase commit** (3PC) protocol exits. This protocol does not block the participants on site failures, except for the case when all sites fail. Thus, the basic conditions that this protocol requires are:

- no network partitioning,

- at least one available site,
- at most *K* failed sites (called *K-resilient).*

Thus, the basic objective of the 3PC is to remove the blocking period for the participants who have voted COMMIT, and are thus, waiting for the global abort/ commit message from the coordinator. This objective is met by the 3PC by adding another phase – the third phase. This phase called **pre-commit** is introduced between the voting phase and the global decision phase.

If a coordinator receives all the commit votes from the participants, it issues a global PRE-COMMIT message for the participants. A participant on receiving the global pre-commit message knows that this transaction is going to commit definitely.

The coordinator on receiving the acknowledgement of PRE-COMMIT message from all the participants issues the global COMMIT. A Global ABORT is still handled the same way as that of in 2PC.

Figure 8 shows the state transition diagram for 3PC. You can refer to further readings for more details on this protocol.

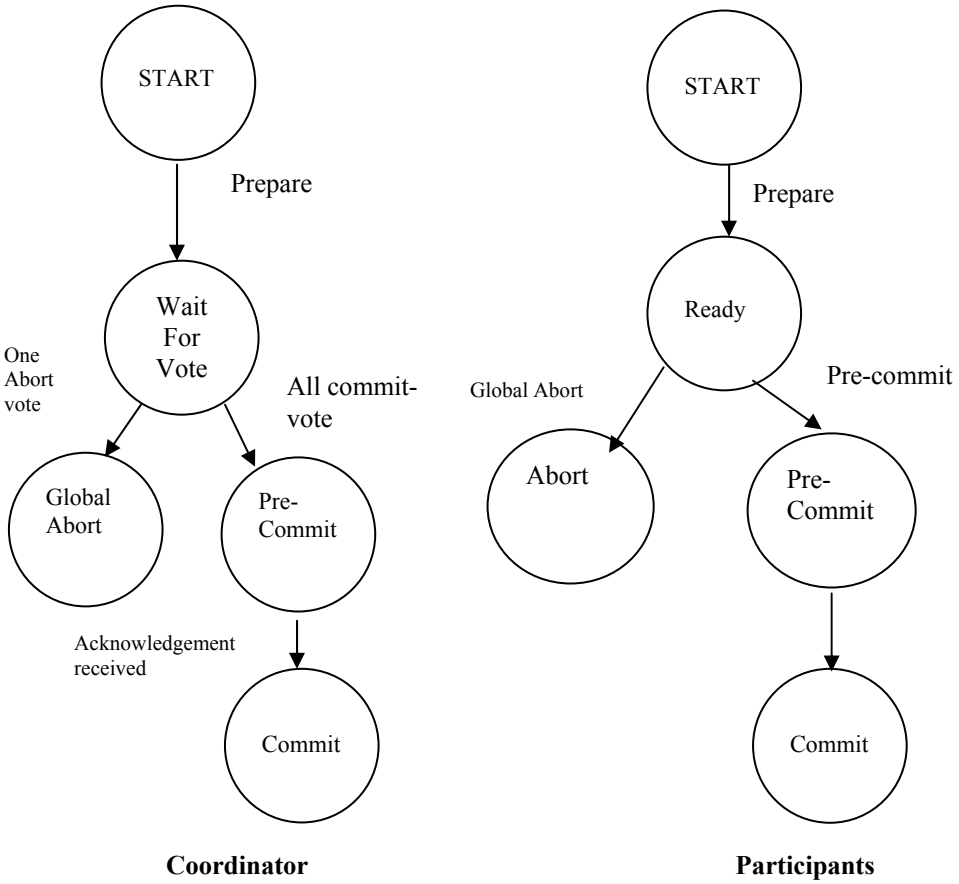


Figure 8: State transition diagram of 3PC

Thus, commit protocols ensure that DDBMS are in consistent state after the execution of a transaction. Transaction is also a unit of recovery. The other issue related to distributed databases is security. We have already discussed recovery and security in centralised systems in this block. A detailed discussion on these topics in the context of distributed database system is beyond the scope of this unit. You may refer to the further readings for more details on distributed database systems.

4.8 CONCEPTS OF REPLICATION SERVER

In many database applications, a large number of remote users may, need to perform transactions or queries that requires large volume of data transfer. This may block the

server. So what can we do about this situation? The solution to this problem may be provided by the concept of the replication server. Data replication distributes data to certain remote sites so that users can access current data as and when they need the data from the site with less traffic. Replication provides a number of benefits. Some of these benefits are improved performance, increased reliability, data availability and automatic recovery.

A Replication Server replicates a database at a remote secondary site. It transmits data from the primary database server (data origination server) to the secondary site on the network. It also handles data transmission to ensure that each replicated database server has an up-to-date copy of the data.

In case the replicated data is updated immediately along with the updation of the source data, then it is referred to as *synchronous replication.* Two possible protocols for this synchronous replication is the 2PC discussed in the previous section. Such replications are useful in applications with critical data such as financial transactions. However, this scheme has some disadvantages as well. For example, a transaction will not be completed if one or more of the sites, holding the replicas, are down. Also the number of messages required to coordinate the synchronisation of data are additional network overheads.

Many commercial distributed DBMSs provide an asynchronous replication in which the target database is updated **after** the source database has been modified. In such a case the delay in regaining consistency may range from a few seconds to several hours or even more. However, the data eventually will synchronise at all replicated sites. This is a practical compromise between data integrity and availability. This form of replication is applicable for organisations that are able to work with replicas that do not necessarily have to be completely synchronised.

Replication Server can do the following operations:

- move transaction to many destinations,
- move data or only a subset of data from one source to another,
- transform data when moving from one source to destination such as merging data from several source databases into one destination database, and
- moving data through a complicated network.

A Replication Server is intended for very close to real-time replication scenarios. The time taken by the data to travel from a source to the destination depends on the number of data items handled by a transaction, the number of concurrent transactions in a particular database, the length of the path (one or more replication servers that the transaction has to pass through to reach the destination), the network bandwidth and traffic etc. Usually, on a LAN, for small transactions, the time taken is about a second.

4.8.1 Replication Server Components

The following are the components of a replication server:

Basic Primary Data Server: It is the source of data where client applications are connected to enter/delete and modify data.

Replication Agent/Log Transfer Manager: This is a separate program/process, which reads transaction log from the source server and transfers them to the replication server.

Replication Server (s): The replication server sends the transactions to the target server, which could be another replication server or the target data server.

Replicate (target) Data server: This is the destination server in which the replication server repeats the transaction that was performed on the primary data server.

4.8.2 Benefits of a Replication Server

The following are the benefits of replication servers:

- Use of Replication Server is efficient, because it only replicates original data that is added, modified, or deleted.
- Replication Servers help in enhancing performance. This is because of the fact that, the Replication Server copies the data to the remote server that a remote user can access over the Local Area Network (LAN).
- Replication server provides an excellent feature for disaster recovery. For example, if the local data server or local network is down and transactions need to be replicated, then the Replication Server will perform all of the necessary synchronisation on the availability of the local data server or the local network.

4.8.3 Functionality

The functionality that a replication server must include is:

- *Scalability:* It should handle the replication of small or large volumes of data.
- *Mapping and transformation:* It should handle replication across heterogeneous DBMSs, which will involve mapping and transformation of the data from one data model or data type into a different data model or data type.
- *Object replication*: The replication server system should allow replicating objects such as logs, indexes, etc. in addition to replication of data.
- *Specification of replication schema:* The system should provide a mechanism that allows authorised users define the data and objects that are to be replicated.
- *Subscription mechanism:* The system should provide a mechanism to allow an authorised user to subscribe to the data and the objects that are to be made available for replication.
- *Initialisation mechanism:* The system should provide a mechanism for replicating the data for the first time.
- *Easy administration:* It should be easy administer the system. It should be easy to check the status and monitor the performance of the replication system components.

4.8.4 Data Ownership

The site that has the privilege to update the data is known as the owner of the data. The main types of ownership are master/slave, workflow, and update-anywhere (also called peer-to-peer or symmetric replication).

Master/Slave Ownership: In such a ownership scheme a site called the Master or Primary site owns the data. Only this site is allowed to modify the data. The other sites can subscribe to the data. They can use the local copy of the data for their purposes. Each site can be a master site for some non-overlapping data sets. The following are some of the examples of the usage of this type of replication:

- *Decision Support System (DSS) Analysis:* Data from one or more distributed databases can be replicated to a separate, local DSS for read-only analysis.
- *Distribution and Dissemination of Centralised Information:* Example of such kinds of application may be a product catalogue and its price. This information could be maintained at the headoffice site and replicated to read-only copies for the remote branch sites.
- *Consolidation of Remote Information:* Data consolidation is a process in which the data that is being updated locally is brought together in a single read-only repository at one location. For example, student details maintained at each

regional office could be replicated to a consolidated read-only copy for the headquarters site.

- *Mobile Computing:* Replication is one of the methods for providing data to a mobile workforce. In this case, the data is downloaded on demand from a local server.

Workflow Ownership: Workflow ownership passes the right to update replicated data from site to site. However, at one time only one site may update the particular data set. A typical example of such ownership may be an order processing system, where for each order processing step like order entry, credit approval, invoicing, shipping, the ownership is passed to a different site. Thus, in this ownership model the application tasks and the rights to update the data for those tasks move.

Update-Anywhere (Symmetric Replication) Ownership: In both the previous ownership types the data can be modified only by one site. However, in many applications we may wish to update data from other sites too. The update-anywhere model creates a peer-to-peer environment that allows different sites equal rights to updating replicated data. Such sharing of ownership may lead to data inconsistencies. Such inconsistencies could require the implementation of advanced protocols for transactions.

☞ Check Your Progress 3

1) What are the steps of two phase commit protocol?

……

……

2) Why do we need replication servers?

……

……

4.9 SUMMARY

This unit provided a brief overview of some of the major concepts used in distributed database management systems. The centralised database systems have to give way to distributed systems for organisations that are more complex and autonomous. The homogenous distributed databases may be a good idea for some organisations. The query processing in DDBMS requires multiple sites in the picture and needs optimisation on the part of data communication cost. Deadlocks detection in a distributed transaction is more complex as it requires local and global wait for graphs. Distributed commit protocols require two phase, one voting phase and second decision-making phase. Sometimes a third phase is added between the two phases called pre-commit state, to avoid blocking. The replication server helps in reducing the load of a database server; it also helps in minimising communication costs. You can refer to more details on these topics in the further readings section.

4.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1)

- Access to remote sites, transfer of queries and data through a network,
- A System catalogue for a distributed system having data distribution details,

- Distributed query processing including query optimisation and remote data access,
- Concurrency control to maintain consistency of replicated sites,
- Proper security control for proper authorisation/access privileges of the distributed data, and
- Recovery services to check the failures of individual sites and communication links.

2) Reference architecture shows data distribution, which is an extension of the ANSI/SPARC three level architecture for a centralised DBMS. The reference architecture consists of the following:

- A set of global external schemas to describe the external view for the entire DBMS,
- A global conceptual schema containing the definition of entities, relationships, constraints and security and integrity information for the entire system,
- A fragmentation and allocation schema, and
- A set of schemas for each local DBMS as per ANSI/SPARC architecture.

Check Your Progress 2

1) A query may be answered by more than one site, in such a case, we would, require that the communication cost among sites should be minimum. You may do data processing at the participant sites to reduce this communication cost.

2) If transactions are distributed at different sites then they should be executed in exactly the same sequence at all the sites.

3) Same set of transactions at different sites may have locked certain resources for which they may be waiting although locally it may seem that T_1 is waiting for T_2 on site 1 and T_2 is waiting for T_1 at site 2, but only on combining these two we know that both transactions cannot proceed. Hence, a deadlock occurs.

Check Your Progress 3

1)

COORDINATOR	PARTICIPANTS
Prepare Wait for vote Collect vote if all COMMIT Send Global COMMIT and ABORT to all participant Waits for acknowledgement from all participant TRANSACTION OVER	Listen to Coordinator Vote Wait for Global Decision Receive decision acts accordingly and sends acknowledgement TRANSACTION OVER

2) They help in:

- High availability,
- Low communication overheads due to local data availability,
- Disaster recovery, and
- Distribution of functions of a system across organisation, etc.