Concurrency Control Protocols

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose.

- Lock-Based Protocols/Two Phase locking Protocols
- Timestamp-Based Protocols
- Multiversion Concurrency Control Techniques
- Multiple Granularity
- Validation-Based Protocols

Lock-based Protocols

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks − A lock on a data item can be in two states; it is either locked or unlocked.

Shared/exclusive: This type of locking mechanism separates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account's balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can't be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the 'write' operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

Lock Compatibility Matrix –

	S	X
S	✔	✘
X	✘	✘

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive(X) on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

- When waiting scheme for locked items is not properly managed
- In the case of resource leak
- The same transaction is selected as a victim repeatedly

Two Phase Locking (2PL) Protocol

Two-Phase locking protocol which is also known as a 2PL protocol. It is also called P2L. In this type of locking protocol, the transaction should acquire a lock after it releases one of its locks.

This locking protocol divides the execution phase of a transaction into three different parts.

- In the first phase, when the transaction begins to execute, it requires permission for the locks it needs.
- The second part is where the transaction obtains all the locks. When a transaction releases its first lock, the third phase starts.
- In this third phase, the transaction cannot demand any new locks. Instead, it only releases the acquired locks.

The Two-Phase Locking protocol allows each transaction to make a lock or unlock request in two steps:

- **Growing Phase**: In this phase transaction may obtain locks but may not release any locks.
- **Shrinking Phase**: In this phase, a transaction may release locks but not obtain any new lock

Let’s see a transaction implementing 2-PL.

	T_1	T_2
1	LOCK-S(A)	
2		LOCK-S(A)
3	LOCK-X(B)	
4	…….	……

5	UNLOCK(A)	
6		LOCK-X(C)
7	UNLOCK(B)	
8		UNLOCK(A)
9		UNLOCK(C)
10	…….	……

This is just a skeleton transaction which shows how unlocking and locking works with 2-PL. Note for:

Transaction T_1:

- Growing Phase is from steps 1-3.
- Shrinking Phase is from steps 5-7.
- Lock Point at 3

Transaction T_2:

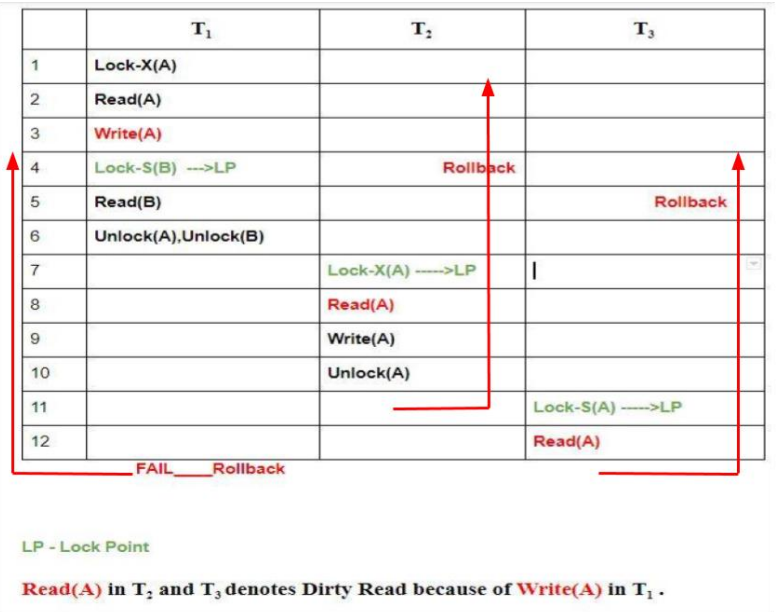
- Growing Phase is from steps 2-6.
- Shrinking Phase is from steps 8-9.
- Lock Point at 6

What is **LOCK POINT ?**The Point at which the growing phase ends, i.e., when transaction takes the final lock it needs to carry on its work. Now look at the schedule, you'll surely understand.

Drawbacks:

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation is possible.

Cascading Rollbacks in 2-PL –
Let's see the following Schedule:



-
- Take a moment to analyze the schedule. Yes, you're correct, because of Dirty Read in T_2 and T_3 in lines 8 and 12 respectively, when T_1 failed we have to rollback others also. Hence **Cascading Rollbacks are possible in 2-PL.** I have taken skeleton schedules as examples because it's easy to understand when it's kept simple. When explained with real time transaction problems with many variables, it becomes very complex.
- **Deadlock in 2-PL –**
  Consider this simple example, it will be easy to understand.Say we have two transactions T_1 and T_2.
- **Schedule:** Lock-X_1(A) Lock-X_2(B) Lock-X_1(B) Lock-X_2(A)
- Drawing the precedence graph, you may detect the loop. So Deadlock is also possible in 2-PL.

  Two-phase locking may also limit the amount of concurrency that occur in a schedule because a Transaction may not be able to release an item after it has used it. This may be because of the protocols and other restrictions we may put on the schedule to ensure serializability, deadlock freedom and other factors. This is the price we have to pay to ensure serializability and other factors, hence it can be considered as a bargain between concurrency and maintaining the ACID properties.

Categories of Two Phase Locking (Strict & Rigorous)

some modifications to 2-PL to improve it. There are three categories:

1. Strict 2-PL
2. Rigorous 2-PL

Now recall the rules followed in Basic 2-PL, over that we make some extra modifications. Let's now see what are the modifications and what drawbacks they solve.

Strict 2-PL –

This requires that in addition to the lock being 2-Phase **all Exclusive(X) Locks** held by the transaction be released until *after* the Transaction Commits.

Rigorous 2-PL –

This requires that in addition to the lock being 2-Phase **all Exclusive(X) and Shared(S) Locks** held by the transaction be released until *after* the Transaction Commits.

Note the difference between Strict 2-PL and Rigorous 2-PL is that Rigorous is more restrictive, it requires both Exclusive and Shared locks to be held until after the Transaction commits and this is what makes the implementation of Rigorous 2-PL more easy.

Timestamp-based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last ‘read and write’ operation was performed on the data item.

Timestamp Ordering Protocol

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction T_i is denoted as $TS(T_i)$.
- Read time-stamp of data-item X is denoted by R-timestamp(X).
- Write time-stamp of data-item X is denoted by W-timestamp(X).

Timestamp ordering protocol works as follows −

- **If a transaction Ti issues a read(X) operation −**
    - If TS(Ti) < W-timestamp(X)
        - Operation rejected.
    - If TS(Ti) >= W-timestamp(X)
        - Operation executed.
    - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation −**
    - If TS(Ti) < R-timestamp(X)
        - Operation rejected.

- If TS(Ti) < W-timestamp(X)
  - Operation rejected and Ti rolled back.
- Otherwise, operation executed.

Thomas' Write Rule

This rule states if TS(Ti) < W-timestamp(X), then the operation is rejected and T_i is rolled back.

Time-stamp ordering rules can be modified to make the schedule view serializable.

Instead of making T_i rolled back, the 'write' operation itself is ignored.

Multiversion Concurrency Control Techniques

Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control**, because several versions (values) of an item are maintained. When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version(s) of the item are retained. Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway—for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a *temporal database* (see Secton 26.2), which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

Several multiversion concurrency control schemes have been proposed. We discuss two schemes here, one based on timestamp ordering and

the other based on 2PL. In addition, the validation concurrency control method (see Section 22.4) also maintains multiple versions.

1. Multiversion Technique Based on Timestamp Ordering

In this method, several versions X_1, X_2, ..., X_k of each data item X are maintained. For *each version*, the value of version X_i and the following two timestamps are kept:

read_TS(X_i). The **read timestamp** of X_i is the largest of all the timestamps of transactions that have successfully read version X_i.

write_TS(X_i). The **write timestamp** of X_i is the timestamp of the transaction that wrote the value of version X_i.

Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version X_i, the value of read_TS(X_i) is set to the larger of the current read_TS(X_i) and TS(T).

To ensure serializability, the following rules are used:

If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).

If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

As we can see in case 2, a read_item(X) is always successful, since it finds the appropriate version X_i to read based on the write_TS of the various existing versions of X. In case 1, however, transaction T may be aborted and rolled back. This happens if T attempts to write a version

of *X* that should have been read by another transaction *T* whose timestamp is read_TS(X_i); however, *T* has already read version X_i, which was written by the transaction with timestamp equal to write_TS(X_i). If this conflict occurs, *T* is rolled back; otherwise, a new version of *X*, written by transaction *T*, is created. Notice that if *T* is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction *T* should not be allowed to commit until after all the transactions that have written some version that *T* has read have committed.

2. Multiversion Two-Phase Locking Using Certify Locks

In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and *certify*, instead of just the two modes (read, write) discussed previously. Hence, the state of LOCK(*X*) for an item *X* can be one of read-locked, write-locked, certify-locked, or unlocked. In the standard locking scheme, with only read and write locks (see Section 22.1.1), a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 22.6(a). An entry of *Yes* means that if a transaction *T* holds the type of lock specified in the column header

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 22.6
Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

on item *X* and if transaction *T* requests the type of lock specified in the row header on the same item *X*, then *T can obtain the lock* because the locking modes are compatible. On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so *T must wait* until *T releases* the lock.

In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions *T* to read an item *X* while a single transaction *T* holds a write lock on *X*. This is accomplished by allowing *two versions* for each item *X*; one version must always have been written by some committed transaction. The second version *X* is created when a transaction *T* acquires a write lock on the item. Other transactions can continue to read the *committed version* of *X* while *T* holds the write lock. Transaction *T* can write the value of *X* as needed, without affecting the value of the committed version *X*. However, once *T* is ready to commit, it must obtain a **certify lock** on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks—which are exclusive locks—are acquired, the committed version *X* of the data item is set to the value of version *X* , version *X* is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure 22.6(b).

In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on *all the items* it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version *X* that was written by a committed transaction. However, deadlocks may occur if upgrading of a read lock to a write lock is allowed, and these must be handled by variations of the techniques discussed in Section 22.1.3.

Validation Based Protocol

Validation based protocol in DBM is a type of concurrency control techniques that works on the validation rules and time-stamps. It is also known as the **optimistic concurrency control technique**. The protocol

associated with three phases for managing concurrent transactions such as read phase, validation phase, and write phase. The optimistic approach of the protocol assumes less interference among concurrent transactions, hence there is no checking process happening while the transactions are executed. This protocol is preferable for short transactions. It uses the local copy of the data for the rollback mechanism that is used to manage rare conflict scenarios and avoids cascading rollbacks.

The Validation based protocol works based upon the following three phases:

- **Read and Execution Phase**: Read phases involve read and execute an operation for Transaction T1. The values of the multiple data items are being read in this phase and the protocol writes the data in a temporary variable. The temporary variable is a local variable that holds the data item instead of writing it to the database.
- **Validation Phase:** The validation phase is an important phase of the concurrency protocol. It involves validating the temporary value with the actual values in the database and to check the view serializability condition.

- **Write Phase:** The write phase ensures valid data to be written to the database that is validated in the validation phase. The protocol performs the rollback operation in case of an invalid scenario of the validation phase.

Various Timestamps Associated

Next, we will discuss various timestamps associated with each phase of the validation protocol. There are three timestamps that control the serializability of the validation based protocol in the database management system, such as.

- **Start(Timestamp)**: The start timestamp is the initial timestamp when the data item being read and executed in the read phase of the validation protocol.
- **Validation(Timestamp):** The validation timestamp is the timestamp when T1 completed the read phase and started with the validation phase.
- **Finish(Timestamp):** The finish timestamp is the timestamp when T1 completes the writing process in the writing phase.

To manage the concurrency between transactions T1 and T2, the validation test process for T1 should validate all the T1 operations should follow TS(T1) < TS(T2) where TS is the timestamp and one of the following condition should be satisfying

Finish T1 < Start T2

- In this condition, T1 completes all the execution processes before the T2 starts the operations.
- It regulates maintaining the serializability.

Start(T2) <Finish(T1) <Validate(T2)

- The validation phase of T2 should occur after the finish phase of T1. This scenario is useful for concurrent transaction serializability.
- The Transactions are able to access the mutually exclusive database resource at a particular timestamp while validating the protocol conditions.

The validation based protocol relies on the timestamp to achieve serializability. The validation phase is the deciding phase for the transaction to be committed or rollback in the database. It works on the equation TS (T1) =Validation (T1) where TS is the time stamp and T1 is the transaction.

Transaction T1	Transaction T2
Read(A)	Read(A)

Read(B) <Validate> Display(B+A)	A=A-40 Read(B) B=B+80 <Validate> Write(A) Write(B)

The transaction table is shown in the example associated with transaction T1 and transaction T2. It represents the schedule produced using validation protocol.

The concurrent transaction process starts with T1 with a reading operation as Read (A) where A is the numeric data element in the database. In the next step, the transaction T2 also reads the same data variable A after some time. Transaction T2 performs an arithmetic operation by subtracting constant value 40 from the variable A. It is represented as A=A-40 in the transaction table. The next step is a read operation on transaction T2 where it's reading another numerical value of variable B as the Read(B). After the read operation completed, the transaction T2 immediately performs an arithmetic operation on the

variable B. It uses the addition operator ‘+’ for adding a constant value as 80 to variable B. The addition operation is represented as B=B+80 in the transaction table.

In the next step of the concurrent transaction, T1 reads the variable B with operation Read (B). Now the validation based protocol comes into the action in the transaction T1 that validates the time stamp of the start phase of T2 lesser than the finishing phase time stamp of Transaction T1 and that is a lesser timestamp as the validate phase of Transaction T2.

Similarly, in the Transaction T2, the validation based protocol validates the timestamps. In the example shown in the table indicates both the validation based protocol is provided with a valid result based on the timestamp condition. And, as the conclusive operations write operations are performed by the transaction T2 using Write (A) and Write (B) statements.

It completes the concurrent transaction scenario with validation based protocol in DBMS.

Advantages

Below are mentioned the advantages:

- The validation based protocol considers high priority to the greater degree of concurrency resulting in fewer conflict scenarios.
- This protocol is known for less number of rollback while maintaining the concurrency control mechanism.

Disadvantages

Below are mentioned some of the disadvantages:

- The validation based protocol may arise in the scenario of transaction starvation. This could be due to the short transaction conflicts associated with this protocol.
- The validation based protocol may not suitable for large transactions as it efficient for maintaining the shorter conflicts in the transactions.

Conclusion

This protocol is implemented in various enterprise systems such as the banking system, ticket booking system, traffic lighting management system where a shared datastore is used for concurrency control. The validation based protocol in DBMS is a mostly used technique that helps to maintain the data consistency, serializability in the multitasking and multi-user environment.

Multiple Granularity

Let's start by understanding the meaning of granularity.

Granularity: It is the size of data item allowed to lock.

Multiple Granularity:

- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area. The higher level database consists of exactly these areas.
- The area consists of children nodes which are known as files. No file can be present in more than one area.
- Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.
- Hence, the levels of the tree starting from the top level are as follows:
    1. Database
    2. Area
    3. File
    4. Record

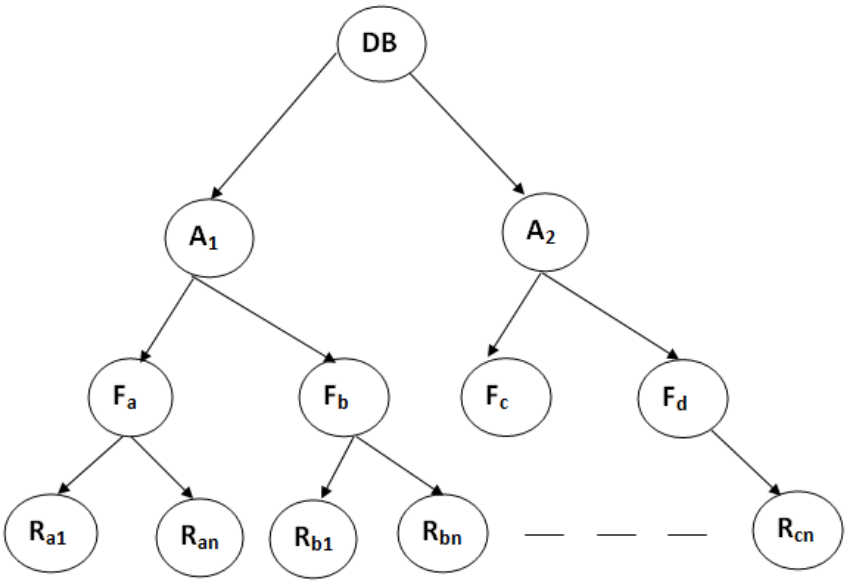


Figure: Multi Granularity tree Hierarchy

In this example, the highest level shows the entire database. The levels below are file, record, and fields.

There are three additional lock modes with multiple granularity:

Intention Mode Lock

Intention-shared (IS): It contains explicit locking at a lower level of the tree but only with shared locks.

Intention-Exclusive (IX): It contains explicit locking at a lower level with exclusive or shared locks.

Shared & Intention-Exclusive (SIX): In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

- If transaction T1 reads record R_{a9} in file F_a, then transaction T1 needs to lock the database, area A_1 and file F_a in IX mode. Finally, it needs to lock R_{a2} in S mode.
- If transaction T2 modifies record R_{a9} in file F_a, then it can do so after locking the database, area A_1 and file F_a in IX mode. Finally, it needs to lock the R_{a9} in X mode.
- If transaction T3 reads all the records in file F_a, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock F_a in S mode.
- If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.

Distributed Database System

A **distributed database** is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.

Features

- Databases in the collection are logically interrelated with each other. Often they represent a single logical database.
- Data is physically stored across multiple sites. Data in each site can be managed by a DBMS independent of the other sites.
- The processors in the sites are connected via a network. They do not have any multiprocessor configuration.

- A distributed database is not a loosely connected file system.
- A distributed database incorporates transaction processing, but it is not synonymous with a transaction processing system.

Distributed Database Management System

A distributed database management system (DDBMS) is a centralized software system that manages a distributed database in a manner as if it were all stored in a single location.

Features

- It is used to create, retrieve, update and delete distributed databases.
- It synchronizes the database periodically and provides access mechanisms by the virtue of which the distribution becomes transparent to the users.
- It ensures that the data modified at any site is universally updated.
- It is used in application areas where large volumes of data are processed and accessed by numerous users simultaneously.
- It is designed for heterogeneous database platforms.
- It maintains confidentiality and data integrity of the databases.

Factors Encouraging DDBMS

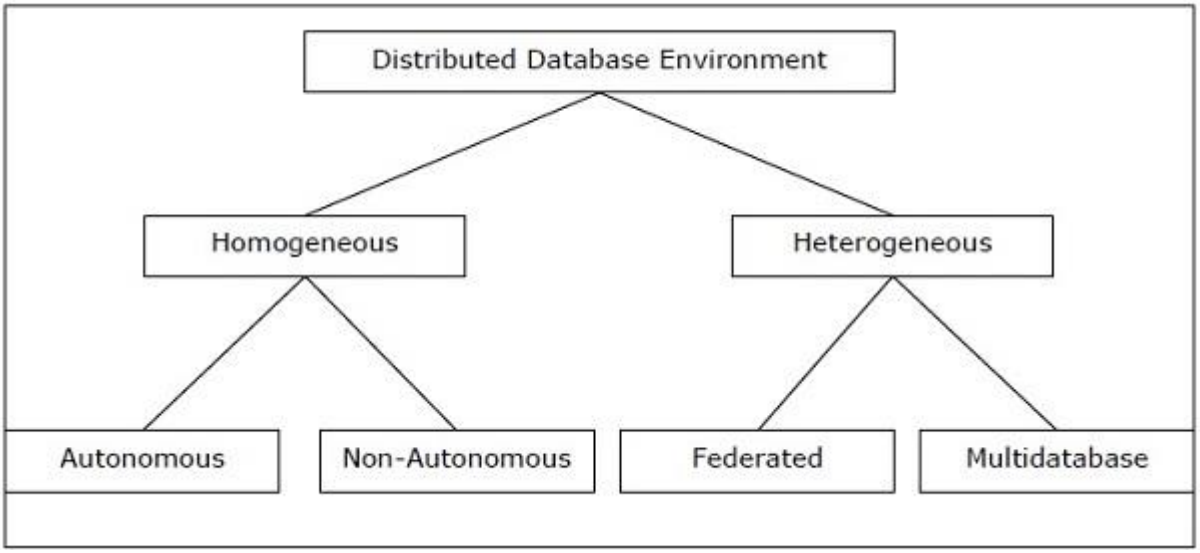
The following factors encourage moving over to DDBMS −

- **Distributed Nature of Organizational Units** − Most organizations in the current times are subdivided into multiple units that are physically distributed over the globe. Each unit requires its own set of local data. Thus, the overall database of the organization becomes distributed.
- **Need for Sharing of Data** − The multiple organizational units often need to communicate with each other and share their data and resources. This demands common databases or replicated databases that should be used in a synchronized manner.
- **Support for Both OLTP and OLAP** − Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) work upon diversified systems which may have common data. Distributed database systems aid both these processing by providing synchronized data.
- **Database Recovery** − One of the common techniques used in DDBMS is replication of data across different sites. Replication of data automatically helps in data recovery if database in any site is damaged. Users can access data from other sites while the damaged site is being reconstructed. Thus, database failure may become almost inconspicuous to users.
- **Support for Multiple Application Software** − Most organizations use a variety of application software each with its specific database support.

DDBMS provides a uniform functionality for using the same data among different platforms.

Types of Distributed Databases

Distributed databases can be broadly classified into homogeneous and heterogeneous distributed database environments, each with further sub-divisions, as shown in the following illustration.



Homogeneous Distributed Databases

In a homogeneous distributed database, all the sites use identical DBMS and operating systems. Its properties are −

- The sites use very similar software.
- The sites use identical DBMS or DBMS from the same vendor.
- Each site is aware of all other sites and cooperates with other sites to process user requests.
- The database is accessed through a single interface as if it is a single database.

Types of Homogeneous Distributed Database

There are two types of homogeneous distributed database −

- **Autonomous** − Each database is independent that functions on its own. They are integrated by a controlling application and use message passing to share data updates.
- **Non-autonomous** − Data is distributed across the homogeneous nodes and a central or master DBMS co-ordinates data updates across the sites.

Heterogeneous Distributed Databases

In a heterogeneous distributed database, different sites have different operating systems, DBMS products and data models. Its properties are −

- Different sites use dissimilar schemas and software.
- The system may be composed of a variety of DBMSs like relational, network, hierarchical or object oriented.
- Query processing is complex due to dissimilar schemas.
- Transaction processing is complex due to dissimilar software.
- A site may not be aware of other sites and so there is limited co-operation in processing user requests.

Types of Heterogeneous Distributed Databases

- **Federated** − The heterogeneous database systems are independent in nature and integrated together so that they function as a single database system.
- **Un-federated** − The database systems employ a central coordinating module through which the databases are accessed.

Data Replication

Data replication is the process of storing separate copies of the database at two or more sites. It is a popular fault tolerance technique of distributed databases.

Advantages of Data Replication

- **Reliability** − In case of failure of any site, the database system continues to work since a copy is available at another site(s).
- **Reduction in Network Load** − Since local copies of data are available, query processing can be done with reduced network usage, particularly during prime hours. Data updating can be done at non-prime hours.
- **Quicker Response** − Availability of local copies of data ensures quick query processing and consequently quick response time.
- **Simpler Transactions** − Transactions require less number of joins of tables located at different sites and minimal coordination across the network. Thus, they become simpler in nature.

Disadvantages of Data Replication

- **Increased Storage Requirements** − Maintaining multiple copies of data is associated with increased storage costs. The storage space required is in multiples of the storage required for a centralized system.
- **Increased Cost and Complexity of Data Updating** − Each time a data item is updated, the update needs to be reflected in all the copies of the data at

the different sites. This requires complex synchronization techniques and protocols.

- **Undesirable Application – Database coupling** − If complex update mechanisms are not used, removing data inconsistency requires complex co-ordination at application level. This results in undesirable application – database coupling.

Some commonly used replication techniques are −

- Snapshot replication
- Near-real-time replication
- Pull replication

Fragmentation

Fragmentation is the task of dividing a table into a set of smaller tables. The subsets of the table are called **fragments**. Fragmentation can be of three types: horizontal, vertical, and hybrid (combination of horizontal and vertical). Horizontal fragmentation can further be classified into two techniques: primary horizontal fragmentation and derived horizontal fragmentation.

Fragmentation should be done in a way so that the original table can be reconstructed from the fragments. This is needed so that the original table can be reconstructed from the fragments whenever required. This requirement is called “reconstructiveness.”

Advantages of Fragmentation

- Since data is stored close to the site of usage, efficiency of the database system is increased.
- Local query optimization techniques are sufficient for most queries since data is locally available.
- Since irrelevant data is not available at the sites, security and privacy of the database system can be maintained.

Disadvantages of Fragmentation

- When data from different fragments are required, the access speeds may be very high.
- In case of recursive fragmentations, the job of reconstruction will need expensive techniques.
- Lack of back-up copies of data in different sites may render the database ineffective in case of failure of a site.

Vertical Fragmentation

In vertical fragmentation, the fields or columns of a table are grouped into fragments. In order to maintain reconstructiveness, each fragment should contain the primary key field(s) of the table. Vertical fragmentation can be used to enforce privacy of data.

For example, let us consider that a University database keeps records of all registered students in a Student table having the following schema.

STUDENT

Regd_No	Name	Course	Address	Semester	Fees	Marks

Now, the fees details are maintained in the accounts section. In this case, the designer will fragment the database as follows −

```
CREATE TABLE STD_FEES AS
   SELECT Regd_No, Fees
   FROM STUDENT;
```

Horizontal Fragmentation

Horizontal fragmentation groups the tuples of a table in accordance to values of one or more fields. Horizontal fragmentation should also confirm to the rule of reconstructiveness. Each horizontal fragment must have all columns of the original base table.

For example, in the student schema, if the details of all students of Computer Science Course needs to be maintained at the School of Computer Science, then the designer will horizontally fragment the database as follows −

```
CREATE COMP_STD AS
   SELECT * FROM STUDENT
   WHERE COURSE = "Computer Science";
```

Hybrid Fragmentation

In hybrid fragmentation, a combination of horizontal and vertical fragmentation techniques are used. This is the most flexible fragmentation technique since it generates fragments with minimal extraneous information. However, reconstruction of the original table is often an expensive task.

Hybrid fragmentation can be done in two alternative ways −

- At first, generate a set of horizontal fragments; then generate vertical fragments from one or more of the horizontal fragments.
- At first, generate a set of vertical fragments; then generate horizontal fragments from one or more of the vertical fragments.

Advantages of Distributed Databases

Following are the advantages of distributed databases over centralized databases.

Modular Development − If the system needs to be expanded to new locations or new units, in centralized database systems, the action requires substantial efforts and disruption in the existing functioning. However, in distributed databases, the work simply requires adding new computers and local data to the new site and finally connecting them to the distributed system, with no interruption in current functions.

More Reliable − In case of database failures, the total system of centralized databases comes to a halt. However, in distributed systems, when a component fails, the functioning of the system continues may be at a reduced performance. Hence DDBMS is more reliable.

Better Response − If data is distributed in an efficient manner, then user requests can be met from local data itself, thus providing faster response. On the other hand, in centralized systems, all queries have to pass through the central computer for processing, which increases the response time.

Lower Communication Cost − In distributed database systems, if data is located locally where it is mostly used, then the communication costs for data manipulation can be minimized. This is not feasible in centralized systems.

Drawbacks of Distributed Databases

Following are some of the adversities associated with distributed databases.

- **Need for complex and expensive software** − DDBMS demands complex and often expensive software to provide data transparency and co-ordination across the several sites.
- **Processing overhead** − Even simple operations may require a large number of communications and additional calculations to provide uniformity in data across the sites.
- **Data integrity** − The need for updating data in multiple sites pose problems of data integrity.
- **Overheads for improper data distribution** − Responsiveness of queries is largely dependent upon proper data distribution. Improper data distribution often leads to very slow response to user requests.