# UNIT I: DISTRIBUTED DATABASES

## SYLLABUS

Distributed Systems – Introduction – Architecture – Distributed Database Concepts –Distributed Data Storage – Distributed Transactions – Commit Protocols – Concurrency Control – Distributed Query Processing

## SNAPSHOT

1. Distributed Systems - Introduction
2. Architecture
   - 2.1 Software Architecture
   - 2.2 System Architecture
3. Distributed Database Concepts
   - 3.1 Transparency
   - 3.2 Autonomy
   - 3.3 Reliability
   - 3.4 Availability
   - 3.5 Consistency
   - 3.6 Idempotency
   - 3.7 Message Persistence
   - 3.8 Distributed Data Storage
   - 3.9 Replication
   - 3.10 Fragmentation
4. Distributed Transactions
   - 4.1 Commit Protocols
   - 4.2 Concurrency Control
   - 4.3 Problems with Concurrent Execution
   - 4.4 Concurrency Control Protocols
5. Distributed Query Processing

## TABLE OF CONTENTS

# DISTRIBUTED SYSTEMS – INTRODUCTION

- In a distributed database system, the database is stored on nodes located at geographically separated sites.
- The nodes in a distributed system communicate with one another through various communication media, such as high-speed private networks or the internet. They do not share main memory or disks.
- Nodes in a distributed database tend to vary more in size and function.
- Sharing data. The major advantage in building a distributed database system is the Provision of an environment where users at one site may be able to access the data Residing at other sites. For instance, in a distributed university system, where each Campus stores data related to that campus, it is possible for a user in one campus to access data in another campus. Without this capability, the transfer of student Records from one campus to another campus would have to rely on some external Mechanism.
- Autonomy. The primary advantage of sharing data by means of data distribution Is that each site can retain a degree of control over data that are stored locally.
- In a homogeneous distributed database system, nodes share a common global schema (although some relations may be stored only at some nodes), all nodes run the same distributed database-management software, and the nodes actively cooperate in processing transactions and queries.
- However, in many cases a distributed database must be constructed by linking together multiple already-existing database systems, each with its own schema and possibly running different database-management software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in query and transaction processing. Such systems are sometimes called federated database systems or heterogeneous distributed database systems.
- In a distributed database system, we differentiate between local and global transactions. A local transaction is one that accesses data only from nodes where the Transaction was initiated. A global transaction, on the other hand, is one that either accesses data in a node different from the one at which the transaction was Initiated, or accesses data in several different nodes.
- Distributed databases have sites that are geographically separated. As a result, the Network connections have lower bandwidth, higher latency, and greater probability of failures, as compared to networks within a single data centre. Systems built on distributed databases therefore need to be aware of network latency, and failures, as well as of physical data location
- Nodes in a distributed database communicate over wide-area networks (WAN). Although wide-area networks have bandwidth much greater than local-area networks, the Bandwidth is usually shared by multiple users/applications and is expensive relative to Local-area network bandwidth. Thus, applications that communicate across wide-area Networks usually have a lower bandwidth. Communication in a WAN must also contend with significant latency: a message May take up to a few hundred milliseconds to be delivered across the world, both due to speed-of-light delays, and due to queuing delays at several routers in the path of the message. Latency in a wide-area setting is a

fundamental problem that cannot be Reduced beyond a point. Thus, applications whose data and computing resources are Distributed geographically must be carefully designed to ensure that latency does not Affect system performance excessively. Wide-area networks also have to contend with network-link failures, a problem that Is relatively rare in local-area networks. Network-link failures may result in Two sites that are both alive having no way to communicate with each other, a situation Referred to as a network partition. In the event of a partition, it may not be possible for A user or an application to access required data. Thus, network partitioning affects the Availability of a system.

# ARCHITECTURE

Distributed system architecture is the classified into two major architectures like as

## SOFTWARE ARCHITECTURE

### LAYERED ARCHITECTURE

- Layer architecture allows to organize the different kinds of components, and every layer make the communication in between its adjacent layers by pushing the requests and obtaining the all response.
- This layered architecture style also separates components into various units then it will be making the sufficient path for communication.
- Then a layer can interact with its neighbouring layers and then further layer for transferring own information to another layer and this process will be continue.
- The request flow path from upper to bottom and response flow towards from bottom to upwards.
- The main benefit of layered architecture is that every layer easily modified individually without getting any affect the entire system.
- This architecture style is usually used for OSI (Open System Interconnection) model.

### DATA-CENTERED ARCHITECTURE

- Data Centred Architecture always contains the common data space at the centre. It has all mandatory data in one area as a shared data space.
- In which, all components are attached with this data and they are getting the follow subscribe/publish kinds of communication.
- For example, Producer-Consumer system, in this system provides the data into common data space and consumers get the request data.

### OBJECT-ORIENTED ARCHITECTURE

- In this object-oriented architecture, all components are acted as objects that transmit the information to each other.
- This architecture style has an arrangement of loosely coupled objects, and these objects can make interaction with each other with helping of this method calls.

- All objects are linked with each other via RPC (Remote Procedure Call) technique or RMI (Remote Method Invocation) technique.
- For examples are Web Services and REST API.

### EVENT-BASED ARCHITECTURE

- This architecture style is usually same as data centre architecture, but just the difference is that event-based architecture contains all events instead of the data.
- All events are available at the centre into Event bus and delivered to needed components whenever required.
- This type of architecture, the whole communication is made via events. Whenever an event occurs, the system as well as the receiver then it gets notify; therefore, data and URL are broadcast with helping of events.
- This system's components are coupled as loosely nature, so it is easy to delete, add and modify them.
- The main advantage of this architecture style, all heterogeneous components make communication with bus by any communication protocol. It has special bus that has ability to manage all kinds of incoming request and process as per the need.
- Event-Based Architecture can support the many communication styles like as:
    - Publisher-subscriber
    - Broadcast
    - Point-to-Point

## SYSTEM ARCHITECTURE

### CLIENT-SERVER ARCHITECTURE

- Client server architecture has two main elements like as client and server.
- The server is capable to execute many tasks such as data processing, computing, and data handling; but whereas the client is whereas user has ability to get access all services and resources which are provided by the server.
- The client makes the requests from the server, but the server will get respond as per respect.
- There is only one server is needed to manage the all activities from remote side. But as per the safe side, we can use multiple servers will get load balancing concepts.
- It is one design feature that the client server architecture contains the centralized security database.
- This database has the all-security information such as credentials and access details.
- Therefore, any user is unable to get log in to the server, without using its security detail. So, it makes this architecture more secure and stable compare the P2P network.
- Key Components:
    - Clients: Clients are the end-user devices or software applications that initiate requests for services or resources. They can be desktop computers, laptops, mobile devices, or even other servers. Clients interact with the user, gather input,

and transmit requests to servers. They rely on servers to provide the requested services and handle the processing.
    - Servers: Servers are dedicated machines or processes responsible for providing services or resources to clients. Servers have specialized software and hardware configurations to handle multiple client connections and perform the requested tasks. They are designed to be robust, scalable, and capable of serving multiple clients simultaneously.
    - Communication: Communication between clients and servers occurs over a network using various protocols. Clients send requests to servers, specifying the desired service or resource. Servers receive these requests, process them, and send back the corresponding responses. Commonly used protocols in client-server architectures include HTTP (Hypertext Transfer Protocol), TCP/IP (Transmission Control Protocol/Internet Protocol), and RPC (Remote Procedure Call).
    - Request-Response Model: The client-server architecture follows a request-response model. Clients send requests to servers, and servers respond with the requested data or perform the requested action. The request contains information such as the type of service required, parameters, and any necessary authentication or authorization details. The server processes the request, performs the necessary computations, accesses databases or other resources, and generates a response, which is then sent back to the client.
    - Centralized Control: In a client-server architecture, servers have a central role in providing services and controlling the system. Clients rely on servers to handle the requested tasks and don't have direct communication or collaboration with each other. Servers manage the system's resources, handle load balancing, enforce security measures, and ensure data consistency.
    - Scalability: Client-server architectures can be designed for scalability by employing techniques like load balancing and server clustering. Load balancing distributes client requests across multiple servers to ensure efficient resource utilization and handle increased workloads. Server clustering involves grouping multiple servers together to work collectively, improving performance and fault tolerance.
    - Security and Authentication: Client-server architectures require robust security measures to protect sensitive data and ensure secure communication. Servers often implement authentication and access control mechanisms to verify the identity of clients before granting access to resources. Encryption protocols like SSL/TLS may be used to secure data transmission between clients and servers.
    - Centralized Data Storage: In many client-server architectures, servers act as data repositories and manage centralized data storage. Clients may send requests to retrieve or modify data stored on servers, which handle the data operations and ensure data consistency.
- Client-server architectures are widely used in various domains, including web applications, database systems, cloud computing, and enterprise systems. They provide a structured and scalable approach to distributed systems, allowing clients to access services and resources efficiently while offloading the processing and storage tasks to dedicated server components.

## PEER-TO-PEER ARCHITECTURE

- In the peer-to-peer network, all "Peers" means all computers which are linked with each other through internet.
- P2P network has not any central server, so each user is capable to share any types of files on any peer over this network. On other words, you can say that every peer on this P2P N/W plays role as server as well as client.
- P2P architecture is mostly suitable for small region area like as house and small office region because in this network every computer plays the role as independent workstation and it can save all data in own hard disk.
- Peer-to-peer (P2P) architectures in distributed systems refer to a decentralized approach where nodes, known as peers, collaborate and share resources directly with each other, without relying on a central authority or server. In P2P systems, each peer acts as both a client and a server, contributing resources and consuming resources from other peers in the network.
- Here are some key characteristics and concepts related to peer-to-peer architectures in distributed systems:
    1. **Decentralization**: P2P architectures are characterized by their decentralized nature. There is no central server or coordinator that controls the system. Instead, peers interact directly with each other to exchange resources and perform tasks.
    2. **Peer Heterogeneity**: Peers in a P2P system can vary in terms of their capabilities, resources, and roles. Some peers may have more computational power, storage capacity, or network bandwidth than others. The system needs to handle this heterogeneity and enable effective resource sharing and collaboration.
    3. **Resource Sharing**: P2P architectures emphasize resource sharing among peers. Resources can include files, processing power, storage space, or any other type of data or functionality. Peers contribute their resources to the network, making them available for other peers to access and utilize.
    4. **Distributed Indexing**: In P2P systems, distributed indexing techniques are used to locate and discover resources available in the network. Peers maintain local indexes or data structures that enable efficient search and retrieval of resources across the network. Examples of distributed indexing algorithms include Distributed Hash Tables (DHTs) like Chord, Kademlia, or CAN (Content Addressable Network).
    5. **Self-Organization**: P2P systems are designed to be self-organizing and self-managing. Peers join and leave the network dynamically, and the system adapts to changes autonomously. Peers may employ algorithms for peer discovery, routing, replication, load balancing, and fault tolerance to ensure the system's stability and efficiency.
    6. **Scalability and Fault Tolerance:** P2P architectures excel in scalability as the system can grow by adding more peers without requiring centralized infrastructure. The decentralized nature of P2P systems also contributes to fault tolerance. If a peer fails or leaves the network, other peers can continue to function without disruption.
    7. **Overlay Networks**: P2P systems often rely on overlay networks to facilitate communication and resource sharing among peers. Overlay networks are virtual networks that are built on top of the underlying physical network infrastructure. They

provide the necessary abstractions and protocols for peers to discover and interact with each other.

8. **Applications**: P2P architectures are commonly used in various applications such as file sharing (e.g., BitTorrent), content distribution networks (CDNs), distributed computing (e.g., SETI@home), blockchain networks (e.g., Bitcoin), and real-time communication (e.g., Skype).

P2P architectures offer advantages such as decentralized control, scalability, fault tolerance, and efficient resource utilization. However, they also pose challenges in terms of security, trust management, data consistency, and ensuring fairness in resource sharing.

Overall, peer-to-peer architectures in distributed systems provide a flexible and dynamic approach to resource sharing and collaboration, enabling nodes to interact directly and distribute tasks and resources among themselves without relying on central authorities or servers.

## FEDERATED ARCHITECTURE:

- In this architecture, each site in the distributed database system maintains its own independent database, but the databases are integrated through a middleware layer that provides a common interface for accessing and querying the data.

- Federated architectures in distributed systems refer to a decentralized approach where multiple autonomous entities or nodes collaborate to perform a task without relying on a central authority or coordinator. In this architecture, each node retains its own local data and computing resources, and the collaboration occurs through communication and coordination among the participating nodes.
- Here are some key characteristics and concepts related to federated architectures in distributed systems:
    1. Autonomy: Each node in the federation has its own autonomy and decision-making capabilities. They can independently perform computations on their local data and decide when and how to participate in the collaborative task.
    2. Local Data Ownership: Each node retains ownership and control over its own local data. Data is not required to be shared or centralized, which can enhance privacy and security.
    3. Communication and Coordination: Nodes communicate with each other to exchange information, synchronize actions, and coordinate the overall task. This communication can occur directly between nodes or through an intermediary, such as a message broker.
    4. Heterogeneity: Nodes in a federated architecture can have different capabilities, resources, and data distributions. They can vary in terms of processing power, memory, network bandwidth, and data volume. The system needs to handle such heterogeneity and enable effective collaboration.
    5. Training and Inference: Federated architectures are often used in machine learning and AI applications where the goal is to collectively train a model using distributed data sources. Nodes perform local model training using their local data and

periodically exchange model parameters or gradients with other nodes for global model aggregation.

6. Privacy and Security: Federated architectures provide a level of privacy and security since the local data remains within the control of individual nodes. Instead of sharing raw data, nodes share only aggregated information or model updates, preserving the privacy of sensitive data.
7. Scalability and Fault Tolerance: Federated architectures can scale well by adding or removing nodes as needed. If a node fails or leaves the federation, other nodes can continue the task without disruption. This fault tolerance is an important aspect of distributed systems.

- Federated architectures are employed in various domains, including distributed machine learning, Internet of Things (IoT) networks, decentralized blockchain networks, and collaborative data analytics. They enable decentralized collaboration while addressing concerns such as data privacy, scalability, and autonomy.

# DISTRIBUTED DATABASE CONCEPTS

## TRANSPARENCY

- The concept of transparency extends the general idea of hiding implementation Details from end users.
- In the case of a traditional centralized database, transparency Simply pertains to logical and physical data independence for application developers. However, in a DDB scenario, the data and software are distributed over multiple sites connected by a computer network, so additional types of transparencies are introduced.
    - Data organization transparency (also known as distribution or network Transparency). This refers to freedom for the user from the operational Details of the network and the placement of the data in the distributed system. It may be divided into location transparency and naming transparency.
        - Location transparency refers to the fact that the command used to perform A task is independent of the location of the data and the location of the node Where the command was issued.
        - Naming transparency implies that once a Name is associated with an object, the named objects can be accessed unambiguously without additional specification as to where the data is located.
    - Replication transparency. Data objects may be stored at multiple sites for better availability, performance, and reliability. Replication transparency makes the user unaware of the existence of these copies.
    - Fragmentation transparency. Two types of fragmentation are possible.
        - Horizontal fragmentation distributes a relation (table) into sub relations that are subsets of the tuples (rows) in the original relation.
        - Vertical fragmentation distributes a relation into sub relations where each sub relation is defined by a subset of the columns of the original relation.

        - Fragmentation transparency makes the user unaware of the existence of fragments.
    - Other transparencies include design transparency and execution transparency referring to freedom from knowing how the distributed database Is designed and where a transaction executes.

## AUTONOMY

- Autonomy determines the extent to which individual nodes or DBs in a connected DDB can operate independently.
- A high degree of autonomy is desirable for Increased flexibility and customized maintenance of an individual node.
- Autonomy Can be applied to design, communication, and execution.
    - Design autonomy refers to independence of data model usage and transaction management technique Among nodes.
    - Communication autonomy determines the extent to which each Node can decide on sharing of information with other nodes.
    - Execution autonomy Refers to independence of users to act as they please.

## RELIABILITY

- Reliability is broadly defined as the probability that a system is running (not down) at a certain time point.

## AVAILABILITY

- Availability is the probability that the system is continuously available during a time interval.
- We can directly relate reliability and availability of the database to the faults, errors, and failures associated with it

## CONSISTENCY

- In a consistent system, all nodes see and return the same information simultaneously. In order to ensure that all nodes have the same data, they need to exchange messages and work in synchronization.
- Generally, the weaker the required level of consistency, the faster the system can work – but at the same time the higher chances that it will not return the latest dataset.

## IDEMPOTENCY

- Idempotency means that the actual event execution will occur only one time regardless the number of times a specific request is executed. By providing a high level of idempotency, developers manage to avoid bad consequences of dropped connections, request errors, and more.

### MESSAGE PERSISTENCE

- When the node which is processing a message goes offline or some other failure happens, there is a risk that a message will be lost.
- Message persistence implies that the message is saved and will be processed after the issue is solved.

## DISTRIBUTED DATA STORAGE

There are 2 ways in which data can be stored on different sites. These are:

### REPLICATION

- In this approach, the entire relationship is stored redundantly at 2 or more sites. If the entire database is available at all sites, it is a fully redundant database.
- Hence, in replication, systems maintain copies of data. This is advantageous as it increases the availability of data at different sites. Also, now query requests can be processed in parallel.
- However, it has certain disadvantages as well. Data needs to be constantly updated. Any change made at one site needs to be recorded at every site that relation is stored or else it may lead to inconsistency. This is a lot of overhead.
- Also, concurrency control becomes way more complex as concurrent access now needs to be checked over several sites.

### FRAGMENTATION

- In this approach, the relations are fragmented (i.e., they are divided into smaller parts) and each of the fragments is stored in different sites where they are required.
- It must be made sure that the fragments are such that they can be used to reconstruct the original relation (i.e., there is not any loss of data).
- Fragmentation is advantageous as it does not create copies of data, consistency is not a problem.

Fragmentation of relations can be done in two ways:

- **Horizontal fragmentation – Splitting by rows –**
  The relation is fragmented into groups of tuples so that each tuple is assigned to at least one fragment.
- **Vertical fragmentation – Splitting by columns –**
  The schema of the relation is divided into smaller schemas. Each fragment must contain a common candidate key to ensure a lossless join.

# DISTRIBUTED TRANSACTIONS

- A distributed transaction is a database transaction in which two or more network hosts are involved. Usually, hosts provide transactional resources, while the transaction manager is responsible for creating and managing a global transaction that encompasses all operations against such resources.
- The transaction coordinator coordinates the execution of the various transactions (both local and global) initiated at that node.
- There are two Types of transaction that we need to consider.
    - The local transactions are those that Access and update data in only one local database;
    - the global transactions are those that Access and update data in several local databases.
- Distributed transactions, as any other transactions, must have all four ACID (atomicity, consistency, isolation, durability) properties
    - Atomicity
        - Transactions are often composed of multiple statements.
        - Atomicity guarantees that each transaction is treated as a single "unit," which either succeeds completely or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged.
        - An atomic system must guarantee atomicity in each situation, including power failures, errors, and crashes.
        - A guarantee of atomicity prevents updates to the database from occurring only partially, which can cause greater problems than rejecting the whole series outright.
        - Consequently, the transaction cannot be observed to be in progress by another database client. At one moment in time, it has not yet happened, and at the next, it has already occurred in whole (or nothing happened if the transaction was cancelled in progress).
    - **Consistency**
        - Consistency ensures that a transaction can only bring the database from one consistent state to another, preserving database invariants:
        - Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.
        - This prevents database corruption by an illegal transaction.
    - Isolation
        - Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time).
        - Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained If the transactions were executed sequentially.

        - Isolation is the main goal of concurrency control; depending on the isolation level used, the effects of an incomplete transaction might not be visible to other transactions.
    - Durability
        - Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash).
        - This usually means that completed transactions (or their effects) are recorded in non-volatile memory. [citation needed]
- A common algorithm for ensuring correct completion of a distributed transaction is the two-phase commit (2PC).

# COMMIT PROTOCOLS

## ONE-PHASE COMMIT PROTOCOL:

- Each slave sends a "DONE" notification to the controlling site once it has successfully finished its transaction locally.
- The slaves await the commanding site's "Commit" or "Abort" message. This period of waiting is known as the window of vulnerability.
- The controlling site decides whether to commit or abort after receiving the "DONE" message from each slave. The commit point is where this happens. It then broadcasts this message to every slave.
- An acknowledgement message is sent to the controlling site by the slave once it commits or aborts in response to this message.

## TWO-PHASE COMMIT

Consider a transaction T initiated at node Ni, where the transaction coordinator Is Ci.

### PHASE-1ST– PREPARE PHASE

- Firstly, the coordinator (Ci) places a log record <Prepare T> on the log record at its site.
- Then, the coordinator (Ci) sends a Prepare T message to all the sites where the transaction(T) executed.
- Transaction manager at each site on receiving this message Prepare T decides whether to commit or abort its component(portion) of T. The site can delay if the component has not yet completed its activity, but must eventually send a response.
- If the site does not want to commit, so it must write on log record <no T>, and local Transaction manager sends a message abort T to Ci.
- If the site wants to commit, it must write on log record <ready T>, and local Transaction manager sends a message ready T to Ci.

### PHASE- 2ND– COMMIT/ABORT PHASE

a) If the coordinator receives ready T from all the participating sites of T, then it decides to commit T. Then, the coordinator writes on its site log record <Commit T> and sends a message commit T to all the sites involved in T.
b) If a site receives a commit T message, it commits the component of T at that site, and write it in log records <Commit T>.
c) If a site receives the message abort T, it aborts T and writes the log record <Abort T>.
d) However, if the coordinator has received abort T from one or more sites, it logs <Abort T> at its site and then sends abort T messages to all sites involved in transaction T.

However, it is possible that some site fails to respond; it may be down, or it has been disconnected by the network. In that case, after a suitable timeout period will be given, after that time it will treat the site as if it had sent abort T.

## THREE-PHASE COMMIT

- Phase one – Obtaining Preliminary Decision
    - It is identical to the 2PC Phase one. Every site must be prepared to make a commitment if directed to do so

Phase 2 of 2PC is divided into Phase Two and Phase Three in 3PC.

- Phase Two –
    - Phase 2 involves the coordinator making a choice like the 2PC (known as the pre-commit decision) and documenting it in several (at least K).
- Phase Three –
    - Phase 3 involves the coordinator notifying all participating sites whether to commit or abort.
    - Under 3PC, despite the coordinator's failure, a choice can be committed using knowledge of pre-commit decisions.

# CONCURRENCY CONTROL

## PROBLEMS WITH CONCURRENT EXECUTION

In a database transaction, the two main operations are READ and WRITE operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent.

So, the following problems occur with the Concurrent Execution of the operations:

## LOST UPDATE PROBLEMS (W – W CONFLICT)

The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.

Consider the below diagram where two transactions TX and TY, are performed on the same account A where the balance of account A is $300.

| Time | $T_x$ | $T_y$ |
|---|---|---|
| $t_1$ | READ (A) | — |
| $t_2$ | A = A - 50 | |
| $t_3$ | — | READ (A) |
| $t_4$ | — | A = A + 100 |
| $t_5$ | — | — |
| $t_6$ | WRITE (A) | — |
| $t_7$ | | WRITE (A) |

**LOST UPDATE PROBLEM**

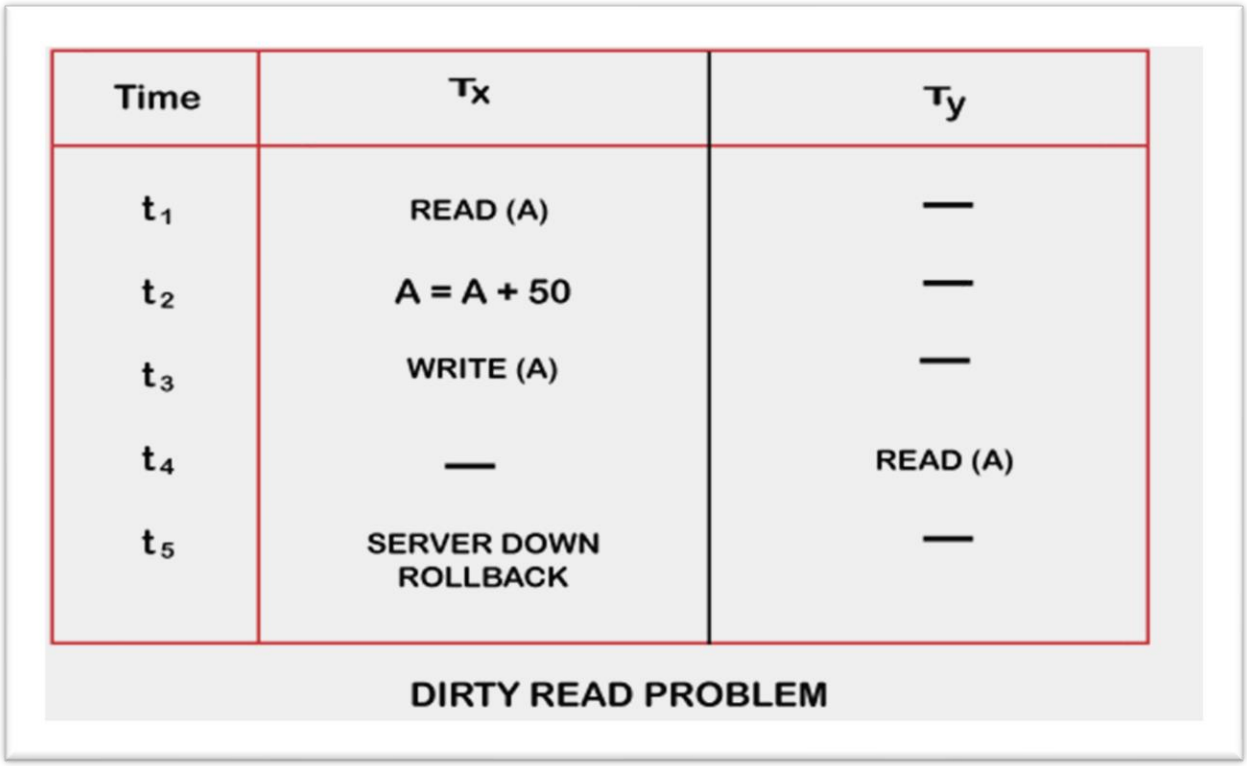
- At time t1, transaction TX reads the value of account A, i.e., $300 (only read).
- At time t2, transaction TX deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction TY reads the value of account A that will be $300 only because TX did not update the value yet.
- At time t4, transaction TY adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction TX writes the value of account A that will be updated as $250 only, as TY did not update the value yet.
- Similarly, at time t7, transaction TY writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by TX is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

## DIRTY READ PROBLEMS (W-R CONFLICT)

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

Consider two transactions TX and TY in the below diagram performing read/write operations on account A where the available balance in account A is $300:




- At time t1, transaction TX reads the value of account A, i.e., $300.
- At time t2, transaction TX adds $50 to account A that becomes $350.
- At time t3, transaction TX writes the updated value in account A, i.e., $350.
- Then at time t4, transaction TY reads account A that will be read as $350.
- Then at time t5, transaction TX rollbacks due to server problem, and the value changes back to $300 (as initially).

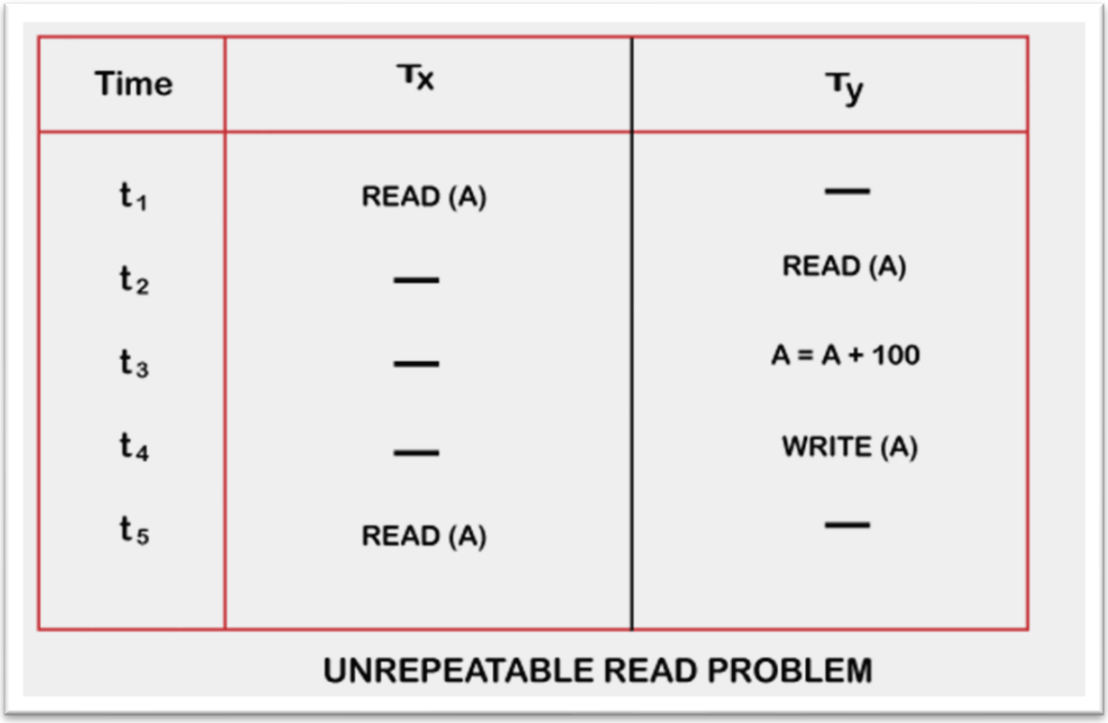
But the value for account A remains $350 for transaction TY as committed, which is the dirty read and therefore known as the Dirty Read Problem.

## UNREPEATABLE READ PROBLEM (W-R CONFLICT)

The unrepeatable read problem, also known as a W-R (Write-Read) conflict, is a data inconsistency issue that can occur in concurrent access scenarios within a database system. It arises when a transaction reads a data item multiple times during its execution, and between

those reads, another transaction modifies or deletes the same data item. As a result, the first transaction obtains different values in its subsequent reads, leading to an inconsistency.

It is also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.




Consider two transactions, TX, and TY, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

- At time t1, transaction TX reads the value from account A, i.e., $300.
- At time t2, transaction TY reads the value from account A, i.e., $300.
- At time t3, transaction TY updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction TY writes the updated value, i.e., $400.
- After that, at time t5, transaction TX reads the available value of account A, and that will be read as $400.

It means that within the same transaction TX, it reads two different values of account A, i.e., $ 300 initially, and after updating made by transaction TY, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

## CONCURRENCY CONTROL PROTOCOLS

### LOCK-BASED PROTOCOL

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

- Shared lock:
  - It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
  - It can be shared between the transactions because when the transaction holds a lock, then it cannot update the data on the data item.
- Exclusive lock:
  - In the exclusive lock, the data item can be both reads as well as written by the transaction.
  - This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

There are four types of lock protocols available:

1. Simplistic lock protocol
   - It is the simplest way of locking the data while transaction.
   - Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it.
   - It will unlock the data item after completing the transaction.
2. Pre-claiming Lock Protocol
   - Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
   - Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
   - If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
   - If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.
3. Two-phase locking (2PL)
   - A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.

There are two phases of 2PL:

- Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.
- Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

- Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
- Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Two-phase Locking is further classified into three types:

a. Strict two-phase locking protocol:
b. Rigorous two-phase locking protocol

c. Conservative two-phase locking protocol

- **Timestamp Ordering Protocol**
  - The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
  - The priority of the older transaction is higher that is why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.

- **Validation Based Protocol**
  - **Read phase**: In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
  - **Validation phase:** In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
  - **Write phase**: If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

## DISTRIBUTED QUERY PROCESSING

A distributed database query is processed in stages as follows:

1. Query Mapping.
   - The input query on distributed data is specified formally Using a query language.
   - It is then translated into an algebraic query on global Relations.
   - This translation is done by referring to the global conceptual Schema.
   - Hence, this translation is largely identical to the one performed in a centralized DBMS. It is first normalized, analysed for semantic errors, Simplified, and finally restructured into an algebraic query.
2. Localization.
   - In a distributed database, fragmentation results in relations Being stored in separate sites, with some fragments possibly being replicated.
   - This stage maps the distributed query on the global schema to separate Queries on individual fragments using data distribution and replication Information.
3. Global Query Optimization.
   - Optimization consists of selecting a strategy from a list of candidates that is closest to optimal.
   - Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs.
4. Local Query Optimization.
   - The local queries are optimized by the local database servers.
   - Finally, the local query results are merged through union operation in case of horizontal fragments and join operation for vertical fragments.

The first three stages discussed above are performed at a central control site, while the last stage is performed locally.

**Example:**

We illustrate this with two simple sample queries. Suppose that the EMPLOYEE and DEPARTMENT relations are distributed at two sites. We will assume in this example that neither relation is fragmented.

**Site 1:**

**EMPLOYEE**

| Fname | Minit | Lname | Ssn | Bdate | Address | Sex | Salary | Super_ssn | Dno |
|---|---|---|---|---|---|---|---|---|---|

10,000 records

**Site 2:**

**DEPARTMENT**

| Dname | Dnumber | Mgr_ssn | Mgr_start_date |
|---|---|---|---|

100 records

The query is submitted at a distinct site 3, which is called the result site because the query result is needed there. Neither the EMPLOYEE nor the DEPARTMENT relations reside at site 3. There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site. The size of the query result is 40 * 10,000 = 400,000 bytes, so 400,000 + 1,000,000 = 1,400,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site. In this case, 400,000 + 3,500 = 403,500 bytes must be transferred. If minimizing the amount o data transfer is our optimization criterion, we should choose strategy 3.