**Distributed Transaction**

### **Two-Phase Commit (2PC) Protocol**

**The 2PC protocol** is a coordination protocol used to ensure that all participating nodes in a distributed transaction either commit the transaction successfully or roll it back, maintaining atomicity.

#### **Phase 1: Prepare Phase (Voting Phase)**

1. **Coordinator Initiates**: The coordinator node sends a PREPARE request to all participant nodes, asking them if they are ready to commit the transaction.
2. **Participants Respond**:
   * **Vote Commit**: If a participant node is ready to commit the transaction (i.e., it has successfully performed all necessary operations and is ready to finalize the changes), it sends a YES vote to the coordinator.
   * **Vote Abort**: If a participant node encounters any issues (e.g., data inconsistency, resource unavailability), it sends a NO vote to the coordinator, indicating it cannot commit the transaction.

#### **Phase 2: Commit/Abort Phase**

**1.Coordinator Decision**:

* + **All YES Votes**: If the coordinator receives YES votes from all participants, it sends a COMMIT request to all participants, instructing them to commit the transaction.
  + **Any NO Votes**: If any participant sends a NO vote, the coordinator sends an ABORT request to all participants, instructing them to roll back any changes made during the transaction.

**2.Participants Execute**:

* + **Commit**: Upon receiving the COMMIT request, participants commit the transaction and release any resources or locks.
  + **Abort**: Upon receiving the ABORT request, participants roll back any changes and release resources.

**3.Final Acknowledgment**: After committing or aborting the transaction, participants send an acknowledgment back to the coordinator.

### **Drawbacks of 2PC**

* **Blocking**: If the coordinator fails after the prepare phase but before sending the final commit/abort command, participants remain in a blocked state, holding resources until the coordinator recovers and issues the final command.
* **No Fault Tolerance**: The failure of the coordinator can leave the system in an uncertain state, as participants do not know whether to commit or abort.

### **Three-Phase Commit (3PC) Protocol**

**The 3PC protocol** is an extension of the 2PC protocol that introduces an additional phase to reduce the blocking problem and improve fault tolerance.

#### **Phase 1: CanCommit Phase (Prepare Phase)**

**1.Coordinator Initiates**: The coordinator sends a CAN\_COMMIT request to all participants, asking if they are prepared to commit the transaction.

**2.Participants Respond**:

* + **Vote Yes**: If a participant is ready, it sends a YES vote to the coordinator.
  + **Vote No**: If a participant is not ready, it sends a NO vote.

#### **Phase 2: PreCommit Phase**

**1.Coordinator Prepares**:

* + If all participants responded with YES, the coordinator sends a PRE\_COMMIT request, signaling them to prepare to commit but not to finalize the transaction yet.
  + If any participant responded with NO, the coordinator sends an ABORT request.

**2.Participants Acknowledge**:

* + Upon receiving the PRE\_COMMIT request, participants acknowledge that they are in a pre-commit state and are ready to finalize the transaction.

#### **Phase 3: Commit/Abort Phase**

**1.Coordinator Decision**:

* + **Commit**: If all participants are ready in the pre-commit phase, the coordinator sends a COMMIT request, instructing participants to finalize the transaction.
  + **Abort**: If an issue arises, or if a failure occurs during the pre-commit phase, the coordinator can send an ABORT request.

**2.Participants Execute**:

* + Participants commit or abort the transaction based on the coordinator’s final command.
  + After executing the final command, participants send an acknowledgment back to the coordinator.

### **Advantages of 3PC over 2PC**

* **Non-blocking**: The additional pre-commit phase allows participants to move to a safe state, reducing the risk of being indefinitely blocked if the coordinator fails.
* **Improved Fault Tolerance**: The protocol is designed to handle certain types of failures more gracefully, avoiding scenarios where participants are left in an uncertain state.

### **Drawbacks of 3PC**

* **Increased Overhead**: The additional phase introduces more communication and processing overhead, which can impact performance.
* **Complexity**: The 3PC protocol is more complex to implement and manage compared to 2PC.

### **Key Concepts of Distributed Transaction Processing**

**1.Distributed Transactions**

* + **Definition**: A distributed transaction is one that spans multiple databases or systems. For instance, a transaction might involve updating records in different databases or services.

**2.Atomicity**

* + **Definition**: The principle that a transaction should be fully completed or fully rolled back. In distributed transactions, this means all parts of the transaction across different systems must succeed, or none should take effect.

**3.Consistency**

* + **Definition**: Ensures that the database remains in a consistent state before and after the transaction. This means all systems involved must be synchronized to reflect the changes made by the transaction.

**4.Isolation**

* + **Definition**: Transactions must be isolated from each other to prevent interference. In a distributed system, isolation ensures that concurrent transactions do not affect each other’s operations across different systems.

**5.Durability**

* + **Definition**: Once a transaction is committed, its changes should persist, even in the case of system failures. This requires mechanisms to ensure that changes are reliably recorded across all involved systems.

**DDBMS Architecture**

### **1. Client-Server Architecture**

In the client-server mode, the system is divided into clients, which request data, and servers, which provide data. The server is typically a centralized database system that handles most of the data processing tasks.

* **Characteristics:**
  + **Client**: The front-end application or user interface that sends requests to the server. Clients are responsible for presenting data to users and may handle some lightweight processing.
  + **Server**: The back-end system that processes queries, manages transactions, and stores data. Servers perform the heavy lifting in terms of data processing and transaction management.
  + **Communication**: Clients and servers communicate over a network. The server processes client requests and sends back the results.
  + **Centralized Data Management**: Although the server might be a single centralized system or a distributed system, it usually manages the core data storage and processing functions.
* **Use Case:** Common in web-based applications where the client is a browser or application, and the server is a powerful database system that processes requests and returns data.

### **2. Peer-to-Peer Architecture**

In a peer-to-peer (P2P) mode, all nodes in the network have equal roles and responsibilities. There is no central server; instead, each node can act as both a client and a server.

* **Characteristics:**
  + **Equality**: Every node has the same capabilities and responsibilities. Any node can initiate a transaction, process data, or store information.
  + **Decentralized Control**: There is no central authority or server. The system is decentralized, meaning each node operates independently but cooperatively.
  + **Resource Sharing**: Nodes share resources (data, processing power) with each other, enabling distributed processing and storage.
  + **Dynamic Network**: Nodes can join or leave the network dynamically, making P2P systems more flexible and resilient to individual node failures.
* **Use Case:** Suitable for distributed file-sharing networks, blockchain systems, and applications that require decentralized control and resource sharing.

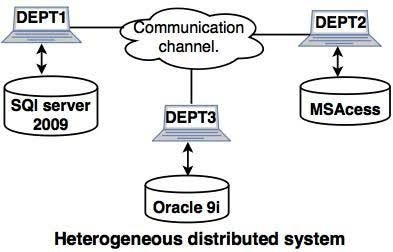
### **3. Multi-Database System (MDBS)**

* A multi-database system is similar to a federated system but typically involves tighter integration. MDBS is designed to work with multiple databases that may not necessarily be autonomous and often requires a global schema to manage **Characteristics**:
* **Global Schema**: Unlike federated systems, an MDBS often uses a global schema or a global view that abstracts the underlying local schemas of the participating databases. This allows users to interact with the system as if it were a single, unified database.
* **Tighter Integration**: MDBS generally requires tighter integration between the databases, which might involve more complex mapping and data transformation processes.
* **Centralized Control**: While the individual databases might retain some autonomy, the MDBS typically exercises more control over how data is accessed and managed across the system.
* **Data Transformation**: MDBS must handle differences in data models, query languages, and transaction management techniques across the different databases, often requiring sophisticated middleware.
* **Use Case**: Ideal for environments where different databases need to be integrated to provide a cohesive data view and facilitate complex queries that span multiple data sources, such as enterprise-wide reporting systems.

**Types of DDBMS**

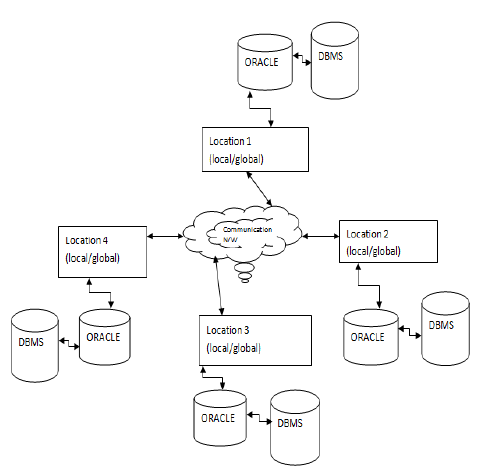
**Heterogeneous Distributed Database**

In a heterogeneous distributed database, different nodes may use different DBMS software, data models, and schemas. This mode requires more complex integration and coordination mechanisms to manage the differences between the systems.



Homogeneous Distributed Database

In a homogeneous distributed database, all the participating nodes (databases) use the same DBMS software and have a uniform data model and schema. This uniformity simplifies the management of the distributed database system.



**STORING DATA IN DDBMS**

### **1. Data Fragmentation**

Data fragmentation is a technique used to break down a database into smaller, more manageable pieces called fragments. These fragments can then be distributed across different nodes in a distributed system. Fragmentation aims to optimize performance, improve data access speed, and ensure that data is stored closer to where it is needed.

#### **a. Horizontal Fragmentation**

* **Definition**: Horizontal fragmentation involves dividing the rows of a table into multiple fragments. Each fragment contains a subset of the rows, typically based on a specific condition or attribute.
* **Example**:
  + Consider a Customers table that includes customer data from different regions.
  + If we horizontally fragment this table based on the Region attribute, we might create fragments like:
    - **Fragment 1**: Contains rows where Region is "North America."
    - **Fragment 2**: Contains rows where Region is "Europe."
    - **Fragment 3**: Contains rows where Region is "Asia."
* **Usage**: Horizontal fragmentation is useful when different nodes in a distributed system handle data from specific regions or subsets of customers. This reduces the amount of data transferred across the network when queries are region-specific.
* **Advantages**:
  + **Improved Query Performance**: Queries run faster on smaller, more targeted fragments.
  + **Localized Data**: Data is stored closer to where it is most frequently accessed, reducing network latency.
* **Challenges**:
  + **Complex Query Processing**: Queries that need to access data across multiple fragments can be complex and require additional processing.
  + **Consistency**: Ensuring consistency across all fragments, especially in the case of updates, can be difficult.

#### **b. Vertical Fragmentation**

* **Definition**: Vertical fragmentation involves dividing the columns of a table into different fragments. Each fragment contains a subset of the columns, along with the primary key to maintain a reference to the original table.
* **Example**:
  + Using the same Customers table, vertical fragmentation might involve creating fragments like:
    - **Fragment 1**: Contains columns CustomerID, Name, and Address.
    - **Fragment 2**: Contains columns CustomerID, PhoneNumber, and Email.
* **Usage**: Vertical fragmentation is useful when different parts of an application or different users require access to only specific columns of a table. This can minimize the amount of data transferred and reduce storage requirements.
* **Advantages**:
  + **Efficient Data Access**: Reduces the amount of data transferred when only specific columns are needed.
  + **Improved Performance**: Smaller fragments mean faster access and processing times.
* **Challenges**:
  + **Reassembly of Data**: When a query requires data from multiple vertical fragments, the system must reassemble the data, which can be complex.
  + **Maintaining Consistency**: Ensuring that updates to a column in one fragment are reflected in all related fragments.

#### **c. Hybrid Fragmentation**

* **Definition**: Hybrid fragmentation is a combination of horizontal and vertical fragmentation. A table is first horizontally fragmented, and then each horizontal fragment is vertically fragmented, or vice versa.
* **Example**:
  + A Sales table could first be horizontally fragmented by Region (creating separate fragments for different regions) and then each regional fragment could be vertically fragmented by ProductCategory.
* **Usage**: Hybrid fragmentation is used in complex systems where both row-based and column-based segmentation are necessary to optimize performance and manageability.
* **Advantages**:
  + **Flexibility**: Allows for highly customized data distribution that can optimize performance for specific query patterns.
  + **Optimized Resource Utilization**: Combines the benefits of both horizontal and vertical fragmentation.
* **Challenges**:
  + **Complexity**: Managing and querying hybrid fragments can be complex and require sophisticated query processing algorithms.

### **2. Data Replication**

Data replication involves creating copies of data and storing them on multiple nodes in a distributed system. Replication increases data availability, reliability, and fault tolerance but also introduces challenges in managing consistency across copies.

#### **a. Full Replication**

* **Definition**: In full replication, every node in the distributed system contains a complete copy of the entire database.
* **Usage**: Full replication is used in systems where high availability and reliability are critical, and where network latency needs to be minimized by ensuring that data is always available locally.
* **Advantages**:
  + **High Availability**: Since every node has a complete copy of the database, the system can continue to function even if several nodes fail.
  + **Fast Query Response**: Queries are processed locally without the need for network communication, leading to faster response times.
* **Challenges**:
  + **Storage Costs**: Storing multiple complete copies of the database can require a significant amount of storage space.
  + **Update Complexity**: Any update to the database must be propagated to all copies, which can be complex and time-consuming, especially in large systems.

#### **b. Partial Replication**

* **Definition**: In partial replication, only a subset of the database is replicated on each node. The subsets chosen are typically based on factors like access frequency or criticality.
* **Usage**: Partial replication is used to balance the trade-offs between storage costs, data availability, and consistency. It’s common in systems where certain data is accessed more frequently or where storage resources are limited.
* **Advantages**:
  + **Efficient Use of Resources**: Reduces the amount of storage required compared to full replication.
  + **Targeted Performance Optimization**: Frequently accessed data is replicated, leading to improved access times for that data.
* **Challenges**:
  + **Data Availability**: If a node that holds a particular fragment fails, that data may become temporarily unavailable.
  + **Consistency Management**: Ensuring that replicated data remains consistent across nodes can be challenging.

### **1. Distributed Concurrency Control**

Concurrency control in distributed databases ensures that transactions are executed concurrently without interfering with each other. In a distributed system, this is more complex because transactions may be spread across multiple nodes, each with its own local database.

#### **a. Locking Protocols**

Locking is a fundamental mechanism used to ensure that only one transaction at a time can modify a piece of data.

* **Two-Phase Locking (2PL)**:
  + **Definition**: In 2PL, a transaction follows two phases: the **growing phase** (where it acquires all the locks it needs) and the **shrinking phase** (where it releases the locks).
  + **Distributed 2PL**: In a distributed setting, a transaction might need to lock data at multiple nodes. The transaction coordinator must ensure that all necessary locks are acquired across the nodes before any operations are performed and that all locks are released only after the transaction commits or aborts.
* **Distributed Deadlock Detection**:
  + **Deadlock**: Occurs when two or more transactions wait for each other to release locks, leading to a cycle of dependencies that prevents them from proceeding.
  + **Detection Techniques**:
    - **Timeout-Based Detection**: Transactions are automatically aborted if they wait for a lock longer than a specified timeout period.
    - **Wait-For Graph (WFG)**: Each node maintains a WFG, and periodically, these graphs are combined across the network to detect cycles, indicating deadlocks.

#### **b. Timestamp Ordering**

Timestamp ordering is a non-locking concurrency control method that assigns a unique timestamp to each transaction, determining the order of execution.

* **Basic Timestamp Ordering**:
  + Transactions are assigned timestamps when they are created.
  + Transactions execute in the order of their timestamps. If a transaction with a later timestamp attempts to access data before an earlier transaction has finished, it is aborted to maintain consistency.
* **Multiversion Timestamp Ordering**:
  + Multiple versions of a data item are maintained, each with its own timestamp.
  + Transactions can access the appropriate version based on their timestamp, reducing conflicts and the need for transaction aborts.
* **Distributed Timestamp Management**:
  + Managing timestamps across a distributed system requires synchronization to ensure that timestamps are consistent and correctly ordered across all nodes. This can be achieved using protocols like Lamport clocks or distributed logical clocks.

#### **c. Optimistic Concurrency Control**

In optimistic concurrency control, transactions execute without restrictions, assuming that conflicts are rare. At the end of the transaction, a validation phase checks for conflicts.

* **Execution Phases**:
  + **Read Phase**: The transaction reads data items and makes tentative changes.
  + **Validation Phase**: Before committing, the transaction is checked for conflicts with other concurrently executing transactions.
  + **Write Phase**: If no conflicts are detected, the transaction commits; otherwise, it is rolled back.
* **Distributed Validation**:
  + In a distributed environment, the validation phase must ensure that no other transaction has modified the data across any node since the transaction began. This often involves communication between nodes to check for conflicts.

#### **d. Distributed Commit Protocols**

Distributed commit protocols ensure that all nodes in a distributed system agree on whether to commit or abort a transaction. The most common protocol is the **Two-Phase Commit (2PC)**:

* **Preparation Phase**: The transaction coordinator sends a "prepare to commit" message to all participating nodes. Each node must decide if it can commit (by checking if it has the necessary locks, etc.).
* **Commit Phase**: If all nodes agree to commit, the coordinator sends a "commit" message. If any node decides to abort, the coordinator sends an "abort" message.

### **2. Distributed Recovery**

Recovery in distributed databases ensures that the system can recover to a consistent state after a failure, whether it's a crash, network partition, or any other type of fault. Recovery mechanisms deal with ensuring that all committed transactions are reflected in the database and that no partial or inconsistent data remains.

#### **a. Logging and Checkpointing**

* **Write-Ahead Logging (WAL)**:
  + **Definition**: Before any changes are made to the database, a log entry describing the change is written to a stable storage (e.g., disk).
  + **Distributed WAL**: Each node in the distributed system maintains its own log. During recovery, these logs must be coordinated to ensure a consistent state across the entire distributed database.
* **Checkpointing**:
  + **Definition**: Periodically, the system takes a snapshot (checkpoint) of the database state, which helps in speeding up recovery.
  + **Distributed Checkpointing**: In distributed systems, coordinated checkpointing ensures that all nodes take a checkpoint at the same logical time, ensuring that a consistent global state is captured.

#### **b. Recovery Protocols**

Recovery protocols help bring the system back to a consistent state after a failure:

* **Undo/Redo Recovery**:
  + **Undo**: Reverses the effects of incomplete transactions.
  + **Redo**: Reapplies the changes of committed transactions that were not fully written to the database.
* **Coordinated Recovery**:
  + After a failure, nodes must coordinate to ensure that all parts of a distributed transaction are either committed or rolled back. This often involves re-running parts of the Two-Phase Commit protocol or checking distributed logs.
* **Quorum-Based Recovery**:
  + In systems using quorum consensus, recovery may involve ensuring that a sufficient number of nodes (a quorum) agree on the state of data before proceeding with recovery. This helps in achieving consistency even in the presence of partial failures.

#### **c. Handling Network Partitions**

* **Network Partitioning**: Occurs when the distributed system is split into isolated sub-networks due to a failure, leading to inconsistent views of the database.
* **Partitioned Recovery**:
  + **Reintegration**: When the network is restored, the system must reconcile any differences in the data state between partitions.
  + **Consistency Models**: Techniques like eventual consistency or strong consistency determine how and when the system reconciles these differences.

#### **d. Failover and Redundancy**

* **Replication-Based Recovery**:
  + In distributed databases with data replication, recovery may involve switching to a replica if the primary node fails. This approach requires careful management to ensure that the replica is up-to-date.
* **Automatic Failover**:
  + Many distributed systems use automatic failover mechanisms, where a backup node automatically takes over if a primary node fails. This minimizes downtime but requires sophisticated coordination to ensure consistency.