24.1

A **DDBMS** manages a database that is stored across multiple locations connected via a network. It allows data to be accessed and updated as if it were centralized.

Key Features:

- **Data Distribution**: Data stored across multiple sites.
- **Transparency**: Users access data without knowing its physical location.
- **Consistency**: Ensures data remains consistent across sites.
- **Scalability**: Easily accommodates additional nodes.
- **Fault Tolerance**: Continues operation despite node failures.

Why Use DDBMS?

1. **Performance**: Faster data access by distributing it closer to users.
2. **Availability**: High reliability due to data replication.
3. **Scalability**: Handles growth without major changes.
4. **Geographic Flexibility**: Supports multi-location data storage.
5. **Cost Efficiency**: Uses smaller, cost-effective systems.

24.2

DDBMS vs. Distributed Processing

A **DDBMS** manages databases spread across multiple sites, focusing on data storage, retrieval, and consistency. It hides the complexity of data location from users and ensures data consistency across all nodes.

In contrast, distributed processing uses multiple computers to handle tasks and computations, focusing on distributing the processing load rather than managing data. It may not provide data consistency or transparency about where processing occurs.

When to Choose DDBMS:

Option for a DDBMS if:

- You need distributed data management with consistency and transparency.
- The application involves data replication for availability and fault tolerance.
- It requires seamless access to large datasets across locations.

24.3

Designing a **centralized relational database** and a **distributed relational database** involves different processes due to their distinct architectures and requirements.

1. **Data Location and Distribution**:
    - In a **centralized database**, all data resides in a single location, simplifying data modeling and design.
    - For a **distributed database**, data must be partitioned and distributed across multiple nodes or sites. This requires additional design steps to decide how data will be **fragmented** (horizontal, vertical, or mixed) and **replicated** (full or partial copies).
2. **Data Consistency**:
    - A centralized database handles consistency easily since there is only one copy of the data.
    - In a distributed database, maintaining **consistency** across multiple sites is challenging. Design must consider strategies like **replication protocols** and **concurrency control** to ensure that data updates are synchronized.
3. **Performance Optimization**:
    - In a centralized database, performance tuning focuses on indexing, query optimization, and storage management within a single server.
    - Distributed databases require additional considerations for **network latency**, **data locality**, and **load balancing** to minimize data transfer and improve query response time.
4. **Fault Tolerance and Availability**:
    - Centralized databases have a single point of failure, making them vulnerable to outages.
    - Distributed databases are designed with **fault tolerance** in mind, requiring redundancy and **replication strategies** to maintain availability even if some nodes fail.
5. **Security**:
    - Centralized databases implement security controls within a single environment, making it easier to manage access.

- In distributed databases, security design must address **data transmission** across networks, access control at multiple sites, and potential vulnerabilities at each node.

24.4

Advantages of a DDBMS

1. **Improved Performance**:
   - Data is distributed closer to users, reducing access time and network latency.
2. **Increased Availability and Reliability**:
   - Data replication across multiple sites ensures that the system can continue operating even if some nodes fail.
3. **Scalability**:
   - A DDBMS can easily scale by adding more nodes without major changes to the system architecture, handling growing data and user demands.
4. **Geographical Flexibility**:
   - Supports organizations with multiple locations, allowing data to be stored locally and accessed globally.
5. **Data Sharing and Integration**:
   - Facilitates data sharing across departments or sites, offering a unified view of data from various sources.

Disadvantages of a DDBMS

1. **Complexity**:
   - Designing, implementing, and maintaining a DDBMS is more complex due to data fragmentation, replication, and synchronization challenges.
2. **Data Consistency Issues**:
   - Ensuring data consistency across distributed nodes is difficult, requiring advanced protocols and mechanisms.
3. **Higher Costs**:
   - Additional hardware, software, and network infrastructure can increase the overall costs of a DDBMS.

4. **Security Risks**:
    - Data transmitted across a network is more vulnerable to breaches, requiring robust security measures at multiple locations.
5. **Potential for Network Overhead**:
    - Data transfer between sites can lead to increased network traffic, impacting performance.

24.5

Homogeneous vs. Heterogeneous DDBMS

Homogeneous DDBMS:

- All nodes use the **same DBMS software** and have a similar data model and schema.
- The entire system appears uniform, simplifying data integration, query processing, and maintenance.
- Data communication and transactions are easier to manage because the systems are consistent.

Heterogeneous DDBMS:

- Nodes use **different DBMS software**, which may have different data models, query languages, or schemas.
- It often involves integrating databases from different vendors (e.g., MySQL, Oracle, SQL Server).
- More complex, requiring middleware or translation layers to handle data differences and ensure interoperability.

Circumstances for Each Type

1. **Homogeneous DDBMS**:
    - Typically arises when an organization plans a distributed database from scratch using a single DBMS vendor.
    - Suitable for environments where data consistency, ease of management, and performance are prioritized.

- Common in scenarios like **global branches of a company** using the same internal database system.

2. **Heterogeneous DDBMS**:
    - Generally arises from **mergers, acquisitions**, or integration of legacy systems, where different databases must work together.
    - Used when an organization needs to **integrate data from different departments** or partners using different database systems.
    - Common in **data integration projects**, like data warehouses or enterprise data hubs.

24.7

Expected Functionality in a DDBMS

A Distributed Database Management System (DDBMS) should provide the following core functionalities:

1. **Data Distribution Management**:
    - Supports data fragmentation, replication, and allocation across multiple sites.
    - Transparently manages the distribution, so users can access data without knowing its physical location.
2. **Data Consistency and Integrity**:
    - Maintains consistency across distributed nodes using replication protocols and concurrency control mechanisms.
    - Enforces data integrity constraints like primary keys, foreign keys, and unique constraints.
3. **Transaction Management**:
    - Provides **ACID (Atomicity, Consistency, Isolation, Durability)** properties for transactions across multiple sites.
    - Supports distributed transactions and ensures that all nodes either commit or roll back changes together.
4. **Query Processing and Optimization**:
    - Enables users to run complex queries across distributed data as if it were a single database.

- Optimizes query execution to minimize network traffic and response time.

5. **Fault Tolerance and Recovery**:
    - Ensures high availability by replicating data across nodes.
    - Supports automatic recovery mechanisms in case of node or network failures.
6. **Security Management**:
    - Implements authentication, authorization, and data encryption to secure data across sites.
    - Controls user access and permissions at multiple locations.
7. **Scalability and Load Balancing**:
    - Efficiently handles increasing data volumes and user requests by scaling out nodes.
    - Balances the load across nodes to ensure optimal performance.
8. **Data Synchronization**:
    - Ensures that changes made at one site are propagated to other relevant nodes to keep data synchronized.
    - Handles potential conflicts during data updates.

24.9

Key Issues in Distributed Database Design

1. **Data Fragmentation**:
    - Deciding how to split data (horizontally, vertically, or mixed) for efficiency without compromising query performance.
2. **Data Replication**:
    - Balancing availability and performance benefits against the complexity of maintaining consistent replicas.
3. **Data Allocation**:
    - Determining the best sites to store data fragments or replicas based on access patterns and costs.
4. **Query Optimization**:

    - Handling complex query execution across multiple sites to minimize data transfer and response time.
5. **Concurrency Control**:
    - Managing simultaneous data updates to prevent conflicts and ensure consistency.
6. **Fault Tolerance**:
    - Ensuring system resilience with data backup, replication, and automatic failover mechanisms.
7. **Security**:
    - Protecting data in transit and at rest, and implementing robust access controls across sites.

Issues with the Global System Catalog (GSC)

- The **GSC** manages metadata about data location, fragmentation, and replication.
- It must handle **consistent updates**, support **efficient query planning**, manage **concurrency**, and ensure **high availability**.
- If the GSC fails or becomes inconsistent, it affects the entire distributed database.

24.10

Horizontal Fragmentation splits a table by rows, where each fragment contains different subsets of data based on conditions (e.g., region). It's useful when queries focus on specific rows, improving data access and performance but requires careful management of data distribution.

Vertical Fragmentation splits a table by columns, where each fragment contains different subsets of attributes. It's beneficial when queries focus on specific columns, reducing I/O and optimizing storage, but reconstructing the table can be complex.

24.11

Alternative Fragmentation Schemes

1. **Horizontal Fragmentation**: Splits data by rows, often based on conditions like region. Useful when queries focus on subsets of rows (e.g., geographic locations).

2. **Vertical Fragmentation**: Splits data by columns. Useful for queries that need specific attributes (e.g., separating customer name and email).
3. **Hybrid Fragmentation**: Combines horizontal and vertical fragmentation for more complex data access patterns.
4. **Range-Based Fragmentation**: Splits data based on value ranges (e.g., transaction dates).
5. **Hash-Based Fragmentation**: Uses a hash function to evenly distribute data across fragments.

Ensuring Correctness During Fragmentation

1. **Preserve Integrity Constraints**: Ensure primary and foreign keys are maintained.
2. **Semantic Consistency**: Data relationships must be intact, and fragments should be joinable.
3. **No Data Loss**: All rows and columns must be represented across fragments.
4. **Reconstruction Validity**: Verify data can be correctly reconstructed from fragments.
5. **Query Testing**: Run sample queries to ensure correct results and performance after fragmentation.

24.12

Differences Between Query Optimization in Centralized vs. Distributed DBMS

1. **Data Location Awareness**:
    - **Centralized DBMS**: Query optimization is simpler because all data is stored in a single location, so the system only needs to optimize access to this centralized data.
    - **Distributed DBMS**: Query optimization is more complex as the system must consider data fragmentation and replication across multiple sites. It needs to decide where to execute the query and how to minimize data transfer across the network.
2. **Query Execution Plan**:
    - **Centralized DBMS**: The query optimizer creates a plan based on local data, considering factors like indexing, join methods, and sorting.

- **Distributed DBMS**: The optimizer must account for data distribution, communication costs, and site availability. It also has to handle distributed joins, ensuring the most efficient data retrieval across nodes.

3. **Cost Model**:
   - **Centralized DBMS**: The cost model focuses mainly on CPU, disk I/O, and memory usage for a single site.
   - **Distributed DBMS**: The cost model becomes more complex as it needs to include network latency, data transfer costs between sites, and the cost of accessing multiple copies of data (replication).
4. **Join Processing**:
   - **Centralized DBMS**: Joins are processed locally, typically using efficient methods like hash join or nested-loop join.
   - **Distributed DBMS**: Distributed joins require data to be fetched from multiple sites, and the optimizer must decide whether to perform the join locally or ship data across the network. Optimizing joins in a distributed system often involves more complex strategies like semi-joins or bloom filters to reduce data movement.
5. **Availability of Resources**:
   - **Centralized DBMS**: The resources (CPU, memory, disk) are available locally, so query optimization is constrained to the capabilities of the single server.
   - **Distributed DBMS**: Resources are spread across multiple sites, and the optimizer must take into account the capacity of each site, which may vary, as well as network bandwidth.

Why Query Optimization in Distributed DBMS is Complex

1. **Data Distribution**: The data is fragmented and possibly replicated across multiple sites, making it challenging to determine where the data resides and how to efficiently access it.
2. **Communication Overhead**: Query optimization must minimize the amount of data transferred between sites. Transmitting large datasets over the network can result in significant performance degradation.
3. **Network Latency**: Latency between distributed nodes introduces delays that must be factored into query planning, unlike in centralized systems where latency is not an issue.
4. **Multiple Copies of Data**: Replication increases complexity as the optimizer must decide whether to access a local copy or a remote one, considering both data freshness and query efficiency.

5. **Concurrency and Load Balancing**: Distributed systems may have varying loads on different sites. The optimizer must take into account the resource utilization of each site to avoid overloading any single node.

24.13

Problems with the central *name server*, which has the responsibility for ensuring uniqueness of all names in the system, are:

- loss of some local autonomy
- performance problems, if the central site becomes a bottleneck
- low availability, if the central site fails, the remaining sites cannot create any new database objects.

An alternative solution is to prefix an object with the identifier of the site that created it. For example, a relation BRANCH created at site S_1 might be named S1.BRANCH. Similarly, we would need to be able to identify each fragment and each of its copies. Thus, copy 2 of fragment 3 of the branch relation created at site S_1 might be referred to as S1.BRANCH.F3.C2. However, this results in loss of distribution transparency.

An approach that resolves the problems with both these solutions uses *aliases* for each database object. Thus, S1.BRANCH.F3.C2 might be known as *local_branch* by the user at site S_1. The DDBMS has the task of mapping aliases to the appropriate database object.