Distributed Database Systems

◆ **Definition:**

A **Distributed Database System (DDBS)** is a collection of **logically interrelated databases** distributed across multiple physical locations that appear to the user as a **single unified system**.

◆ **1. Key Features of Distributed Databases:**

- **Data Distribution:** Data is stored across multiple sites or nodes.
- **Location Transparency:** Users are unaware of the physical location of data.
- **Concurrency Control:** Ensures correctness during simultaneous access.
- **Fault Tolerance:** Handles node failures without loss of service.
- **Autonomy:** Each site can manage its own database independently.

◆ **2. Architectures of Distributed Database Systems**

🧩 **A. Client-Server Architecture**

- **Client** sends queries.
- **Server** processes and returns data.
- Simple, centralized control over distribution.

🧩 **B. Peer-to-Peer Architecture**

- Each node is both a client and a server.
- Nodes collaborate for query execution.
- Better fault tolerance and scalability.

🧩 **C. Multi-Database Architecture (Federated)**

- Each database is managed independently.
- Integration occurs at a higher layer.
- Suitable when databases differ in types or platforms.

🧩 **D. Cloud-Based DDBS**

- Hosted on cloud infrastructure (e.g., AWS, Azure, GCP).
- Elastic, globally distributed, pay-per-use.
- Examples: **Google Spanner**, **Amazon Aurora**, **CockroachDB**.

◆ **3. Types of Data Distribution:**

Type	Description
Fragmentation	Dividing a table into smaller pieces (horizontally/vertically).
Replication	Copying data across multiple sites to improve availability.
Allocation	Determining where to place fragments and replicas for performance.

🔸 4. Consistency Models in Distributed Databases

Consistency models define how updates to data are seen by different users and systems.

Model	Description
Strong Consistency	All nodes see the same data at the same time.
Eventual Consistency	Updates will propagate over time, and all nodes will eventually see the same data.
Causal Consistency	Writes that are causally related must be seen in order.
Session Consistency	Guarantees consistency within a single user session.
Read-Your-Writes	Ensures a user sees the results of their own writes.
Quorum-Based Consistency	Requires a majority of nodes to agree on a value (used in systems like DynamoDB).

🔸 5. Advantages of Distributed Databases:

- Improved **reliability** and **availability**
- Better **performance** via local access
- **Scalability** across regions or data centers
- **Modular growth** – nodes can be added easily

🔸 6. Challenges:

- **Network latency** and partitioning
- **Distributed transaction management**
- Maintaining **consistency** across replicas
- **Security** across distributed infrastructure

🔸 7. Real-World Examples:

System	Type	Used By
Google Spanner	Globally distributed SQL	Google, Cloud customers
Amazon DynamoDB	NoSQL, eventual consistency	AWS services, real-time apps
Apache Cassandra	Peer-to-peer, NoSQL	Netflix, Instagram
MongoDB Atlas	Document-based	Web and mobile apps

CAP Theorem (Brewer's Theorem)

🔹 **Definition:**

The **CAP theorem** states that in any **distributed data system**, it is **impossible to simultaneously guarantee all three** of the following properties:

CAP Component	Explanation
Consistency (C)	Every read receives the most recent write or an error.
Availability (A)	Every request receives a (non-error) response, without guarantee of recent data.
Partition Tolerance (P)	The system continues to function even when communication between nodes is lost.

🔹 **Implication:**

You can **only achieve two of the three** at any time:

System Type	Guarantees	Example
CP	Consistency + Partition Tolerance	HBase, MongoDB (in some configs)
CA	Consistency + Availability	Rare in distributed systems
AP	Availability + Partition Tolerance	DynamoDB, Couchbase, Cassandra

---

2. ACID Properties (Relational Databases)

🔹 **Definition:**

ACID stands for a set of **guarantees** that traditional databases provide to ensure data integrity during transactions.

Property	Meaning
Atomicity	All operations in a transaction are completed; if one fails, all are rolled back.

Property	Meaning
Consistency	The database moves from one valid state to another, maintaining integrity constraints.
Isolation	Concurrent transactions do not interfere with each other.
Durability	Once a transaction is committed, it remains so even in the event of a failure.

🔹 **Example:**

A banking transaction that transfers ₹100 from Account A to B:

- **Atomicity**: Both debit and credit occur, or neither.
- **Consistency**: Balances are correctly updated.
- **Isolation**: Other operations do not interfere.
- **Durability**: Changes persist even if the system crashes after the commit.

---

3. Eventual Consistency (Common in NoSQL Systems)

🔹 **Definition:**

Eventual consistency means that, **given enough time**, **all nodes in a distributed system will converge to the same data value**, assuming no new updates are made.

🔹 **Characteristics:**

- Prioritizes **availability** over immediate consistency.
- Used in systems with **high scalability** needs and **loose latency requirements**.

🔹 **Real-World Example:**

- **Amazon DynamoDB** and **Cassandra**:
  - A write may appear immediately on one node but take time to propagate to others.
  - Eventually, all replicas will agree.

🔹 **Benefits:**

- **Low latency** responses.
- **High availability** during network partitions.
- **Scalability** across multiple regions.

🔹 **Drawback:**

- Risk of **temporary stale reads** or conflicts if multiple writes happen at the same time.

Comparison Table:

Aspect	ACID	CAP Theorem	Eventual Consistency
Focus	Transaction integrity	Distributed system limitations	Long-term consistency of replicas
Trade-offs	Performance and scalability	Must choose 2 of 3 (C, A, P)	Sacrifices consistency temporarily
Used In	Relational DBs (SQL)	Distributed databases & systems	NoSQL DBs (DynamoDB, Cassandra)
Suitability	Banking, Finance, ERP	All large-scale systems	Social media, E-commerce, IoT