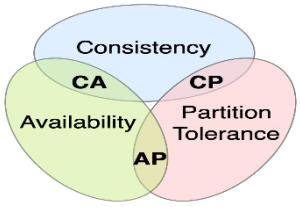
1. Explain the CAP theorem referred during the development of any distributed application.

1.2 CAP Theorem

The **CAP Theorem** (also called **Brewer's Theorem**) states that a **distributed database system** can guarantee **only two out of the following three properties at the same time**:

- **Consistency (C)**
- **Availability (A)**
- **Partition Tolerance (P)**

In real distributed systems, **network failures (partitions) are unavoidable**, so a system must choose between **Consistency** or **Availability** when a partition occurs.



---

1. Consistency (C)

All nodes in the system show the **same data at the same time**.

- After any update, **every read shows the latest value**.
- Ensures **accuracy**, but may reduce performance.

Example: Banking systems — the balance must be the same on all branches/devices.

---

2. Availability (A)

Every request gets a **response** (success or failure), even if some nodes are down.

- The system remains **operational at all times**.
- Data may be slightly **outdated but accessible**.

Example: E-commerce websites — product pages must load even if some servers face issues.

---

3. Partition Tolerance (P)

The system continues to operate **even when there is a network failure or communication break** between servers.

- Essential for **large-scale distributed systems** across regions.
- Cannot avoid partitions in real-world cloud systems.

Example: Cloud data centers where connectivity issues may temporarily occur.

Real-World Example — Netflix

Netflix uses AP systems (Availability + Partition Tolerance) for video streaming.

- If one server or region goes down, streaming still works from another server → **high availability**.
- But non-critical data (like "recently watched") may update with a delay → **weaker consistency**.

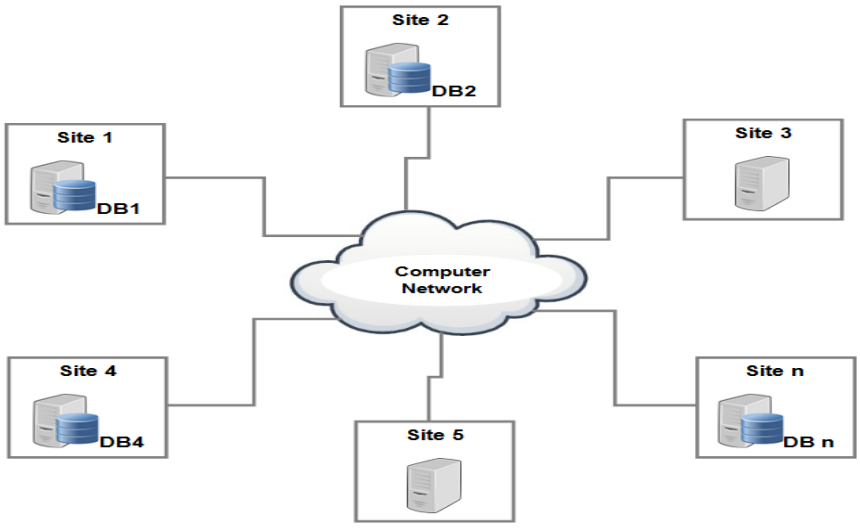
---

2. Draw and explain architecture of Distributed database system. State the reasons for building distributed database systems

1.1 Distributed Database System (DDBS)

A **Distributed Database System (DDBS)** is a type of database system where **data is stored across multiple sites or locations**, but to the user, it appears as a **single unified database**.

Each site has its **own local database** and is connected to other sites through a **communication network**.



Key Points from the Diagram

- Multiple sites (Site 1, Site 2, Site 3, Site 4) — each with its **own local DB**.
- Sites are **interconnected** using a communication network.
- Network allows **data sharing**, **coordination**, and **message passing**.
- Users can **access and manage data without knowing its physical location**, as the system appears as **one central database**.

Architecture of Distributed Database System

1. Sites (Nodes)

- Each site has its **own DBMS**.
- Handles **local transactions** independently.

2. Databases

- Each site keeps a **local database** (subset or replica of full data).
- Data may be:
    - **Fragmented** – split across sites.
    - **Replicated** – copies stored at multiple sites.

3. Communication Network

- Connects all sites.
- Supports:
    - Data exchange
    - Query routing
    - Transaction coordination
    - Message passing

4. Distributed DBMS (DDBMS) Software

- Manages all distributed databases.
- Responsible for:
    - Data consistency
    - Query processing
    - Replication
    - Failure recovery

Working of the Architecture

1. A user sends a **query or transaction**.
2. The **DDBMS** identifies where the required data resides.
3. The **communication network** routes the request to the correct site.
4. Each site processes its part **locally**.
5. Results from all sites are **combined** and returned as a **single response**.

Advantages of DDBMS

1. **High reliability** – failure of one site doesn't stop the system.
2. **Improved performance** – local processing is faster.
3. **Scalability** – easy to add new sites.
4. **Data sharing** – globally accessible data.

Disadvantages of DDBMS

1. **Complex design and management**.
2. **Data consistency issues** due to replication delays.
3. **High communication cost** between sites.
4. **Security challenges** in networked locations.

Example

A multinational company stores:

- Employee records → **India**
- Payroll data → **USA**
- Project data → **Germany**

Yet all departments can access and query data as a **single unified database**.

---

3. Explain structured, Semi-structured and Unstructured data types with examples.

2.1 Structured Data

Structured Data refers to data that is **organized in a fixed format**, usually in **rows and columns**, making it easy to **store, query, and analyze** using relational databases.

Key Characteristics

- Follows a **predefined schema** (data types, relationships, constraints).
- Stored data in the form of **row** and **coloums**(table format)
- Easily accessed using **SQL (Structured Query Language)**.
- Ensures **data consistency, accuracy, and integrity**.

Example Table

Student_ID	Name	Age	Department
101	Ayush	20	CSE
102	Nisha	21	IT

Each **column has a defined data type** and each **row represents one record**.

Common Databases

- Oracle
- MySQL
- MS SQL Server
- DB2

2.2 Unstructured Data

Unstructured Data refers to information that **does not follow a fixed format or predefined schema**.
It **cannot be stored in rows and columns** like relational databases and often contains **rich, complex content** such as text, images, videos, audio, etc.

Key Characteristics

- **No fixed structure** or data model.
- May be **text-heavy or multimedia-based**.

- Hard to analyze using **traditional SQL**.
- Requires **special tools** (Hadoop, NoSQL) for storage and processing.

Examples

- Text documents (PDFs, Word files)
- Images & videos (JPEG, MP4)
- Audio files (MP3, podcasts)
- Emails, social media posts, chat logs

Storage Systems

- **NoSQL databases:** MongoDB, Cassandra
- **Big Data systems:** Hadoop, Data Lakes
- **Cloud storage:** AWS S3, Google Cloud Storage

2.3 Semi-Structured Data

Semi-Structured Data lies **between structured and unstructured data**.

It **does not follow a rigid schema** like relational data but still includes **organizational elements** (tags, key-value pairs) that make it easier to analyze.

Key Characteristics

- **Flexible schema** — structure can change over time.
- **Self-describing format** — tags/attributes describe data.
- Easier to store and query than unstructured data.
- Commonly used in **web applications and APIs**.

Examples

JSON

```
{ "name": "Akshay", "age": 20, "department": "CSE" }
```

XML

```
<student>
  <name>Akshay</name>
  <age>20</age>
  <department>CSE</department>
</student>
```

Common Storage Systems

- **NoSQL databases:** MongoDB, CouchDB
- **Data formats:** JSON, XML, CSV

---

4. Explain how NOSQL databases are different than relational databases?Describe in detail the column NOSQL data model with example.

NoSQL Database

A **NoSQL (Not Only SQL) Database** is a type of database that **stores and manages data without using a fixed schema or traditional table-based structure**.

It is designed to handle **large volumes of unstructured or semi-structured data**, offering:

- **High scalability**
- **High flexibility**
- **Fast performance**

NoSQL is widely used in **modern web**, **cloud applications**, **big data processing**, and **real-time applications**.

Need of NoSQL Database

✓ 1. Traditional RDBMS cannot handle modern requirements

- Not suitable for **big data**, **real-time web apps**, or **rapidly growing datasets**.

✓ 2. Data today is mostly unstructured

(Images, JSON, logs, social media, IoT data, etc.)

✓ 3. Applications require high performance

- Need **high availability**, **scalability**, and **faster response time**.

✓ 4. Distributed environments require horizontal scaling

- Cloud and distributed computing need databases that can scale easily across servers.

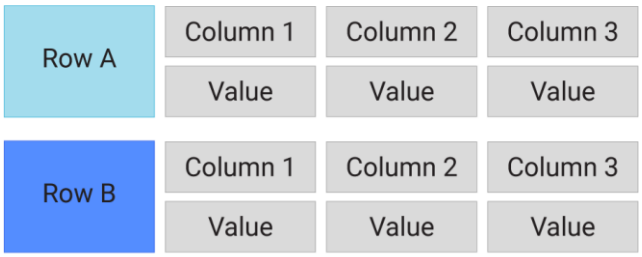
✓ 5. Schema flexibility

- NoSQL allows data to be stored in **varying**, **changing**, or **diverse formats** without strict schema rules.

4.4 Wide Column Stores

A **Wide Column Store** (also called a **Column-Oriented Database**) is a type of **NoSQL database** that stores data in **columns instead of rows**.

This structure is highly efficient for **large-scale analytical and read-intensive workloads**.



How it works

- Each **column** stores values of a particular attribute
  (e.g., Name, Age, City).
- Related columns are grouped into a **column family**.
- Only the **required columns** are read from disk → faster performance.
- Good for analytics, logs, and big data workloads.

Example Table

UserID	Name	Age	City
1	Ritvik	20	Mumbai
2	Nisha	21	Delhi
3	Ashish	22	Pune

In a wide column store, **each column (Name, Age, City) is stored separately**, not row-by-row.

Key Features

1. **Column-oriented storage** — ideal for analytical queries.
2. **Flexible schema** — each row may have different columns.
3. **Column families** — group related data logically.
4. **Highly scalable & distributed** — handles massive datasets across servers.

Advantages

- **Fast query performance** for analytical operations.
- **Efficient storage** (compresses similar column data).
- **Highly scalable** → used in big data systems.
- **Flexible schema** → columns may vary between rows.

Disadvantages

- **Complex setup and management**.
- Not ideal for **transactional (OLTP) systems**.
- **Limited query language support** compared to SQL.

Examples of Wide Column Databases

- **Apache Cassandra**
- **HBase**
- **ScyllaDB**

---

5. Enlist the different types of NOSQL databases and explain with suitable examples.

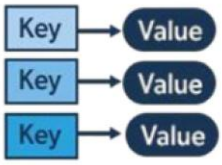
Types of NoSQL Databases

NoSQL databases are classified into **four main types**, based on how they store and retrieve data:

1. **Document-based databases**
2. **Key-value stores**
3. **Column-oriented databases**
4. **Graph-based databases**

4.1 Key-Value Store

A **Key-Value Store** is a NoSQL database that stores data as a **collection of key–value pairs**.



- **Key** = unique identifier
- **Value** = actual data (text, number, JSON, object, etc.)

It works like a dictionary or map in programming.

Example

Key	Value
"Name"	"Akash"
"Age"	20
"Department"	"CSE"

Key Features

1. **Simple structure** — just key–value pairs, no relationships
2. **High performance** — fast lookups using keys
3. **Schema-less** — values can be any type
4. **Distributed design** — scales across servers

Advantages

- Extremely **fast read/write** operations
- Easy **horizontal scaling**
- Ideal for **caching** and **session storage**

Disadvantages

- Cannot handle **complex queries**
- Limited ability to model **relationships**

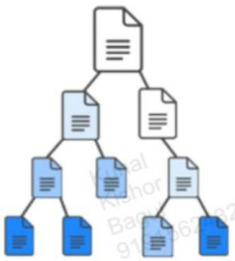
Examples of Key-Value Databases

- **Redis**
- **Amazon DynamoDB**
- **Riak**

- **Berkeley DB**

4.2 Document Store

A **Document Store** stores and manages data as **documents**, usually in formats like **JSON, BSON, or XML**.



Documents can include:

- Key-value pairs
- Arrays
- Nested objects

This allows storing complex, hierarchical data.

Example (JSON Document)

```
{
  "student_id": 101,
  "name": "Arpit",
  "department": "CSE",
  "courses": [
    { "course_name": "DBMS", "credits": 3 },
    { "course_name": "Computer Graphics", "credits": 4 }
  ]
}
```

All related data is kept in **one document**, unlike relational tables.

Key Features

1. **Schema-less** — each document can be different
2. **Hierarchical data representation**
3. **Easy horizontal scalability**
4. **Querying based on documents**, not SQL

Advantages

- Flexible structure — easy to modify
- Faster access — related data stored together
- Natural for developers (JSON-like structure)

Disadvantages

- Data duplication may happen
- No joins (relationships not supported like RDBMS)
- Inconsistency risk if schema changes frequently

Examples of Document Databases

- **MongoDB**
- **CouchDB**
- **RavenDB**

4.3 Graph Database

A **Graph Database** stores data using:

- **Nodes** (entities)
- **Edges** (relationships)

- **Properties** (attributes)

It is designed to model **highly connected data**, such as social networks, recommendation engines, and fraud detection systems.

Example

Social network:

- **Nodes** → Riya, Vishal, Akash
- **Edges** →
    - Riya follows Vishal

- Vishal follows Akash

Graph easily represents paths like:
Riya → Vishal → Akash

Key Features

1. **Node-based structure** — stores interconnected objects
2. **Relationship-focused** — edges stored as first-class elements
3. **Fast traversal** — ideal for queries over relationships
4. **Schema-less** — flexible, new relationships can be added anytime

Advantages

- Best for **relationship-heavy data**
- Faster querying of connected nodes
- Intuitive representation of complex structures

Disadvantages

- Not suitable for table-like data
- Limited scalability for very large datasets
- Complex queries may be slow over huge graphs

Examples of Graph Databases

- **Neo4j**
- **OrientDB**
- **Amazon Neptune**

---

6. Compare SQL and NOSQL Database.

Aspect	RDBMS (Relational Database Management System)	NoSQL (Not Only SQL Database)
Data Model	Based on **tables** (rows and columns)	Based on **key-value**, **document**, **column**, or **graph** models
Schema	**Fixed schema** — structure must be defined before data insertion	**Schema-less** — structure can change anytime
Scalability	**Vertically scalable** (add more power to a single server)	**Horizontally scalable** (add more servers to distribute data)
Query Language	Uses **SQL** (Structured Query Language)	Uses **non-SQL queries** (JSON, APIs, etc.)
Consistency	Follows **ACID** properties (strict consistency)	Follows **BASE** properties (eventual consistency)
Data Type	Best for **structured data**	Handles **unstructured**, **semi-structured**, and **structured data**
Relationships	Supports **relations and joins** between tables	Usually **does not support joins**, uses **embedding or linking**
Transactions	Supports **multi-row, multi-table transactions**	Supports **simple, single-record operations** efficiently
Performance	**Slower** for large-scale or unstructured data	**Faster** for large-scale, distributed systems
Storage System	**Centralized**	**Distributed** across multiple nodes
Examples	MySQL, Oracle, PostgreSQL, SQL Server	MongoDB, Cassandra, CouchDB, Neo4j

7. Explain BASE properties with its significance. How soft state of system is depending on Eventual consistency property?

4.5 BASE Properties

The **BASE Properties** describe the characteristics of **NoSQL databases**, which focus on **availability and scalability** rather than strict consistency (like in ACID of SQL databases).

BASE stands for:

- **Basically Available (BA)**
- **Soft State (S)**
- **Eventually Consistent (E)**

---

1. Basically Available (BA)

- The system **guarantees availability** — every request receives a response (success or failure).
- Even if some nodes fail, the system remains operational using **replication** or **redundancy**.
- Focus is on **keeping the system always accessible**.

Example:
An e-commerce website still loads product pages even if one server is down.

---

2. Soft State (S)

- The system's state **may change over time**, even without user input.
- Data updates **propagate gradually** across nodes.
- Temporary inconsistencies are **allowed** until synchronization completes.

Example:
A product's stock value may show differently on different servers for a short time until all replicas update.

---

3. Eventually Consistent (E)

- The system does **not guarantee immediate consistency**.
- But **all nodes become consistent over time** once updates are propagated.
- Eventually, all replicas will hold the **same data**.

Example:
If a user changes their profile photo, it may take a few seconds before it updates on all devices.

Significance of BASE Properties

1. High Availability

- System remains **operational even if some nodes fail**.
- Users can still access data without interruption.

2. Improved Scalability

- BASE enables **data distribution** across multiple servers (sharding).
- Nodes can be added/removed easily without affecting performance.

3. Better Performance

- Relaxing strict consistency makes **reads/writes faster**.
- Improves **response time** in real-time apps.

4. Fault Tolerance

- Supports **replication and recovery** during network/server failures.
- Data remains accessible even during disruptions.

5. Flexibility with Data Models

- Handles **unstructured and semi-structured data** (JSON, XML).
- Easily adapts to changing application needs without redesigning schema.

How soft state of system is depending on Eventual consistency property?

Soft State depends on Eventual Consistency because temporary inconsistencies occur while updates propagate across nodes. During this time, the system's state keeps changing automatically until all replicas become consistent. Thus, Soft State exists only because consistency is achieved eventually, not immediately.

---

8. Explain the CRUD operations used in MongoDB with example.

CRUD Operations

CRUD stands for the four basic operations performed on data in any database:

C – Create

R – Read

U – Update

D – Delete

These operations are used in both **SQL** and **NoSQL** databases like MongoDB.

---

1. Create (C)

Used to **insert new documents** into a collection.

Syntax

db.collection_name.insertOne({ field1: value1, field2: value2, ... })

db.collection_name.insertMany([{...}, {...}])

Example

db.students.insertOne({

  name: "Tanish",

  age: 20,

  department: "CSE"

})

✓ Inserts a new student record into the *students* collection.

---

2. Read (R)

Used to **retrieve documents** from a collection.

Syntax

db.collection_name.find(query, projection)

- **query** → filter conditions
- **projection** → selects which fields to show (optional)

Example

db.students.find({ department: "CSE" })

✓ Displays all students belonging to the CSE department.

---

3. Update (U)

Used to **modify existing documents** in a collection.

Syntax

db.collection_name.updateOne(filter, { $set: { field: value } })

db.collection_name.updateMany(filter, { $set: { field: value } })

Example

```
db.students.updateOne(
   { name: "Raju" },
   { $set: { age: 21 } }
)
```

✓ Updates the age of the student named Raju to 21.

---

4. Delete (D)

Used to **remove documents** from a collection.

Syntax

db.collection_name.deleteOne(filter)

db.collection_name.deleteMany(filter)

Example

db.students.deleteOne({ name: "Shivani" })

✓ Deletes the document where **name = Shivani**.

---

9. Describe the following operations with MongoDB syntax: [8]
i) Map-Reduce ii) Aggregation pipeline

Aggregation

Aggregation in MongoDB is a powerful feature used to **process, analyze, and transform data** inside collections.

It supports operations like:

- Filtering
- Grouping
- Sorting
- Calculations

It works similar to the **GROUP BY** clause in SQL.

MongoDB provides the **aggregation framework** using the:

aggregate()method, which works as a **pipeline**.

Concept: Aggregation Pipeline

An **aggregation pipeline** is a sequence of stages where:

- Each stage performs **one operation** on the data.
- The **output of one stage** becomes the **input to the next stage**.

Example pipeline:

Filter → Group → Sort → Output

Syntax

```
db.collection_name.aggregate([
  { stage1: { ... } },
  { stage2: { ... } },
  ...
])
```

Example

Students collection:

```
[
  { "name": "Tushar", "department": "CSE", "marks": 85 },
  { "name": "Rahul",  "department": "CSE", "marks": 90 },
  { "name": "Priya",  "department": "IT",  "marks": 80 }
]
```

Goal: Find average marks in each department

```
db.students.aggregate([
  {
    $group: {
```

```
    _id: "$department",
    average_marks: { $avg: "$marks" }
   }
 }
])
```

Output

```
[
 { "_id": "CSE", "average_marks": 87.5 },
 { "_id": "IT",  "average_marks": 80 }
]
```

5.4 MapReduce

MapReduce is a data processing and aggregation technique used to handle **large volumes of data**, especially in MongoDB, Hadoop, and other big data systems.

It divides tasks into **two phases**:

1. **Map phase**
2. **Reduce phase**

MapReduce runs **in parallel** across multiple servers.

✓ Map Phase

- Processes each document.
- Extracts required information.
- Emits data as **key–value pairs**.

✓ Reduce Phase

- Takes output from the Map phase.
- Groups values by key.
- Performs operations like **sum**, **count**, or **average**.
- Produces final summarized results.

Example of MapReduce

Students collection:

```
[
  { "name": "Ayush", "department": "CSE", "marks": 85 },
  { "name": "Priya", "department": "CSE", "marks": 90 },
  { "name": "Shivani", "department": "IT", "marks": 80 }
]
```

Map Function

```
var mapFunction = function() {
  emit(this.department, this.marks);
};
```

Reduce Function

```
var reduceFunction = function(key, values) {
  return Array.avg(values);
};
```

MapReduce Command

```
db.students.mapReduce(
  mapFunction,
  reduceFunction,
  { out: "avg_marks" }
)
```

Output (stored in avg_marks collection)

```
[
  { "_id": "CSE", "value": 87.5 },
  { "_id": "IT",  "value": 80 }
]
```
