# UNIT 5 :- NoSQL Databases

***Centralized Database System:***

A centralized database system is a type of database system where all the data is stored, managed, and maintained in a single central location, typically on a single server or mainframe. Users access the database from various remote locations, but the actual database resides in one place.

***Key Characteristics:***

- Single location: All data is stored in one physical location.
- Controlled access: Central control over data integrity, security, and backup.
- Easier to manage: Easier to administer and update because everything is in one place.
- Risk of failure: If the central system fails, the entire database becomes unavailable.

***Examples of Centralized Database Systems:***

1. **Library Database System**

A university or public library where all book inventory, borrower records, and catalog information are stored on one central server.

2. **Banking System (Legacy Models)**

Older banking systems where all branch data was managed from a central data center.

3. **Enterprise Resource Planning (ERP) Systems**

Centralized ERPs like SAP or Oracle E-Business Suite in a traditional setup.

4. **Government Databases**

National identity databases (e.g., central citizen ID systems).

5. **Hospital Information Systems**

A hospital network with one central server storing all patient records, test results, and billing information.

## 1. i. Introduction to Distributed Database System (DDBS)

### 🔹 Definition:

A **Distributed Database System** is a type of database system in which:

- **Data is stored across multiple physical locations** (computers or sites),
- But it appears to users as **a single unified database**.

Each site (computer/server) is connected via a **network** and manages its **own local database**.

---

### 🔹 Example:

Let's say a company like **Amazon** has:

- Server in **Delhi** storing North India customer data,
- Server in **Mumbai** storing West India customer data,
- Server in **Chennai** storing South India customer data.

Still, an Amazon employee can access data from all regions as if it were one single database — this is made possible by a **Distributed Database System**.

---

## ii. How It Works

A **Distributed DBMS (DDBMS)** controls and coordinates data across different sites.

It performs:

1. **Data Distribution** → Decides where to store data.
2. **Replication** → Keeps copies of data for faster access and fault tolerance.
3. **Transparency** → Hides the complexity of multiple databases from the user (so the system "feels" centralized).

---

## iii. Architecture of Distributed Database

There are mainly **two architectures**:

- **(A) Homogeneous DDBS**
  - All sites use the **same DBMS software** and data models.
  - Easy to manage and integrate.
  - Example: All sites using **Oracle DB**.
- **(B) Heterogeneous DDBS**
  - Sites may use **different DBMS software** (e.g., Oracle, MySQL, SQL Server).
  - Data models and query languages may differ.
  - Example: One branch uses **Oracle**, another uses **MySQL**.

---

## IV. Types of Data Distribution

**(i) Data Fragmentation**

Data is **divided into smaller pieces (fragments)** and stored in different locations.

- **Horizontal Fragmentation** → Rows divided across sites
  Example: Customers in Delhi stored in Delhi server, Customers in Mumbai stored in Mumbai server.
- **Vertical Fragmentation** → Columns divided across sites
  Example: Personal details stored in one site, transaction details stored in another.

**(ii) Replication**

Same data is **copied** and stored at multiple sites to improve availability and performance.

**(iii) Mixed (Hybrid)**

Combination of fragmentation and replication.

## V. Characteristics of DDBS

1. **Transparency**
    - Users don't know data is distributed.
    - Types:
        - *Location transparency* (User doesn't know where data is stored)
        - *Replication transparency*
        - *Fragmentation transparency*
2. **Reliability and Availability**
    - If one site fails, others can still operate.
3. **Scalability**
    - Easy to add new sites or databases.
4. **Resource Sharing**
    - Data and resources can be shared among sites.

---

## VI. Advantages of Distributed Database System

| Advantage | Explanation | Example |
|---|---|---|
| **1. Improved Reliability** | Failure at one site does not affect the entire system. | If the Chennai server fails, Delhi and Mumbai servers still work. |
| **2. Faster Query Response** | Data is stored near users → faster access. | Delhi users access Delhi server quickly. |
| **3. Scalability** | Easy to add new sites without major changes. | Adding a new Hyderabad branch database. |

| Advantage | Explanation | Example |
|---|---|---|
| **4. Data Sharing** | Different departments/branches can share data easily. | HR and Finance access the same employee database. |
| **5. Local Autonomy** | Each site controls its own database operations. | Each branch can manage its local customers independently. |
| **6. Reduced Communication Cost** | Local queries use local databases. | Local office queries don't always travel to central server. |

---

## VII. Disadvantages of Distributed Database System

| Disadvantage | Explanation | Example |
|---|---|---|
| **1. Complex System Design** | More difficult to design and maintain. | Synchronizing data between many sites is hard. |
| **2. Data Integrity Issues** | Ensuring all copies are updated correctly is challenging. | If one site updates data and others don't, inconsistency occurs. |
| **3. Higher Cost** | Requires powerful hardware, network, and skilled DBAs. | Setup and maintenance are costly. |
| **4. Security Challenges** | Data across networks is more vulnerable. | Need strong encryption and authentication. |
| **5. Difficult Backup and Recovery** | Coordinating recovery from multiple sites is complex. | If multiple sites fail, recovery is tough. |

| Disadvantage | Explanation | Example |
|---|---|---|
| **6. Communication Overhead** | Network traffic may increase due to data transfer between sites. | Queries involving multiple servers cause delays. |

## VIII. . Real-World Examples

| Example | Description |
|---|---|
| **Banking Systems** | Banks maintain data of different branches at different locations, yet customers can access data from any branch. |
| **E-commerce (e.g., Amazon, Flipkart)** | Product, order, and customer data are distributed globally for fast access. |
| **Airline Reservation Systems** | Distributed databases manage flight bookings from various locations. |
| **University Databases** | Different departments have local databases connected to a university-wide system. |

## IX. COMPARISON

| Aspect | Centralized DB | Distributed DB |
|---|---|---|
| **Data Storage** | Single location | Multiple locations |
| **Reliability** | Failure affects entire system | High reliability |
| **Performance** | Slower for remote users | Faster local access |
| **Complexity** | Easy to design | Complex management |
| **Cost** | Cheaper | Expensive |

## X. CONCLUSIONS

A **Distributed Database System** provides:

- **Faster, more reliable, and scalable** data management across multiple locations
  But it also requires **careful design, synchronization, and security management**.

In short:

"A Distributed Database is the backbone of modern global systems — balancing performance and complexity."

---

# 2. What is CAP Theorem?

**CAP Theorem** is a fundamental concept in **Distributed Database Systems**, proposed by **Eric Brewer** in 2000.

It states that:

In any distributed database system, it is **impossible** to guarantee all **three properties** — **Consistency**, **Availability**, and **Partition Tolerance** — at the same time.

So, at any moment, a distributed database can **satisfy only two** of the three.

---

## I. The Three Properties of CAP

| Property | Meaning | Example |
|---|---|---|
| **C – Consistency** | Every read receives the **most recent write** or an **error**. | When one user updates a record, all others immediately see that update. |
| **A – Availability** | Every request (read/write) gets a **response**, even if some nodes fail. | System always replies — may return old data but never stops responding. |

| Property | Meaning | Example |
| --- | --- | --- |
| **P – Partition Tolerance** | The system continues to work even if **network communication fails** between nodes. | If a network link between servers breaks, the system still runs independently. |

---

**Simplified Meaning**

- **Consistency:** All users see the same data, everywhere.
- **Availability:** The system is always up and responsive.
- **Partition Tolerance:** The system survives network failures.

---

## II. Why We Can't Have All Three

In a **distributed system**, servers communicate over a network.
If the network breaks (**partition occurs**), the system must **choose** between:

- **Consistency (C):** Stop serving data until nodes are synchronized (data always correct, but slower).
- **Availability (A):** Keep serving data even if some are outdated (system stays fast, but may return old data).

Therefore, **only two properties can coexist** at a time.

---

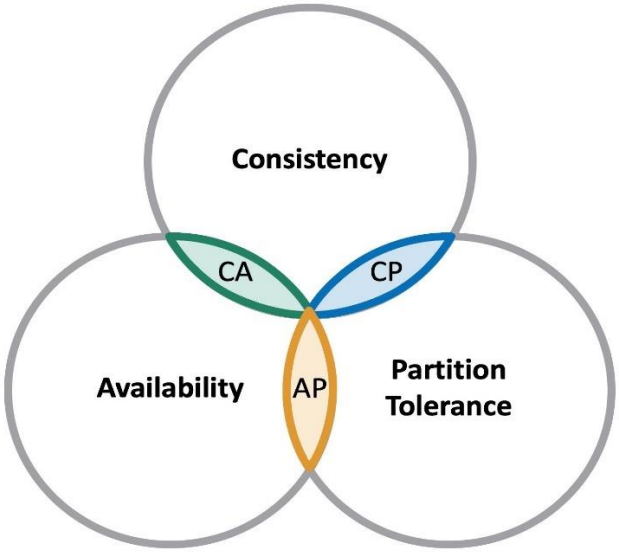
## III. CAP Theorem (Conceptual Diagram)

HERE consider three concepts:-

Consistency (C)

Availability (A)

Partition Tolerance (P)




**Circle diagram for cap theorem**

At any given time, a distributed system can only be in one of these regions:

- **CA** (Consistency + Availability)
- **CP** (Consistency + Partition Tolerance)
- **AP** (Availability + Partition Tolerance)

## IV. CAP Theorem Combinations with Examples

| Type | Properties | Example System | Explanation |
|---|---|---|---|
| **CA (Consistency + Availability)** | System always gives accurate data and is always available — but fails if network partition happens. | Traditional **RDBMS** like MySQL in a single location. | Works well in centralized systems, not across networks. |
| **CP (Consistency +** | Ensures accuracy and tolerates network failure — but may | **HBase**, **MongoDB (configurable)**, **Google Bigtable** | If nodes can't talk, some writes/reads are blocked to |

| Type | Properties | Example System | Explanation |
|---|---|---|---|
| **Partition Tolerance)** | not be available during partition. | | maintain consistency. |
| **AP (Availability + Partition Tolerance)** | Always responds and tolerates network failures — but data may be temporarily inconsistent. | **Cassandra**, **CouchDB**, **Amazon DynamoDB** | Allows different nodes to update independently; sync happens later. |

## V. Real-World Example

**Example 1 — Banking System (CP System):**

A banking database must always show **correct account balance**.

- If a network link breaks, it may **delay requests** instead of showing incorrect data.
- Prioritizes **Consistency** and **Partition tolerance** over **Availability**.

**Example 2 — E-commerce System (AP System):**

When you buy a product on Amazon,

- You should still be able to browse and add to cart even if some servers are disconnected.
- The system chooses **Availability** and **Partition tolerance**, sacrificing strict **Consistency** (you may briefly see outdated stock info).

## VI. Summary Table

| Property | Description | If Network Fails |
|---|---|---|
| **Consistency (C)** | All nodes have same up-to-date data | Some requests may be delayed |

| Property | Description | If Network Fails |
|---|---|---|
| **Availability (A)** | Always responds to requests | May return old data |
| **Partition Tolerance (P)** | Keeps running even if nodes can't communicate | May sacrifice either C or A |

---

### VII. CAP in Modern Distributed Databases

| Database | CAP Category |
|---|---|
| **MongoDB** | CP (Consistency + Partition Tolerance) |
| **Cassandra** | AP (Availability + Partition Tolerance) |
| **DynamoDB** | AP |
| **HBase / Bigtable** | CP |
| **MySQL (single node)** | CA |

---

### VIII. Key Takeaways

- The **CAP theorem** explains why distributed systems must **trade off** between **consistency**, **availability**, and **partition tolerance**.
- **Partition tolerance** is **essential** (since networks can always fail), so in practice, designers must choose **between C and A**.
- There is **no perfect system** — it depends on application requirements:
  - **Banking:** prefer **CP**
  - **Social Media:** prefer **AP**

---

## 10. In Short

## CAP Theorem says:

“In a distributed system, you can’t have Consistency, Availability, and Partition Tolerance all at once — you must choose any two.

## WHAT IS JSON?

- **JSON** stands for **JavaScript Object Notation**.
- It is a **lightweight data-interchange format** that is **easy for humans to read and write**, and **easy for machines to parse and generate**.

### Key Features of JSON:

- Lightweight and fast
- Uses key/value pairs
- Structured using objects {} and arrays []
- Often used in web APIs and configuration files
- Language-independent, but based on JavaScript syntax

### JSON Example:

```
{
  "name": "John Doe",
  "age": 30,
  "email": "john@example.com",
  "isStudent": false,
  "skills": ["HTML", "CSS", "JavaScript"]
}
```

### WHAT IS XML?

**XML** stands for **eXtensible Markup Language**.
It is a **markup language** designed to **store and transport data**, with a focus on **self-descriptive structure**.

### Key Features of XML:

- Uses custom tags to describe data
- Hierarchical structure with opening and closing tags
- More verbose than JSON
- Commonly used in older web services, RSS feeds, and config files
- Language-independent

- **XML Example:**

```
<person>
  <name>John Doe</name>
  <age>30</age>
  <email>john@example.com</email>
  <isStudent>false</isStudent>
  <skills>
    <skill>HTML</skill>
    <skill>CSS</skill>
    <skill>JavaScript</skill>
  </skills>
</person>
```

## When to Use What?

- Use **JSON** when working with **modern web apps, REST APIs, or mobile apps**.
- Use **XML** when dealing with **older systems, document-based data, or SOAP APIs**.

- **COMPARISON BETWEEN JSON & XML**

| Feature | JSON | XML |
|---|---|---|
| Format | Key-value pairs | Tags with nested elements |
| Readability | More human-readable | More verbose |
| Data Types | Supports strings, numbers, booleans, arrays | Only string values (by default) |
| Parsing | Faster and simpler | Slower and more complex |
| Use Case | Modern APIs, configs | Legacy systems, data exchange |

# Types of Data:

- **Structured, Unstructured Data and Semi-Structured Data.**

In **Database Management Systems (DBMS)** or **Big Data Systems**, data comes in different **forms and formats**.
Based on how **organized** and **easily storable** the data is, it is classified into three main types:

1. **Structured Data**
2. **Unstructured Data**
3. **Semi-Structured Data**

## 1. Structured Data

- **Definition:**

Structured data is **highly organized** and stored in **fixed fields** (like tables with rows and columns).
It follows a **defined schema**, meaning every record has the same format and type of information.

## Key Characteristics:

- Stored in **Relational Databases (RDBMS)** such as MySQL, Oracle, SQL Server.
- Easily searchable using **SQL queries**.
- Data fits neatly into **tables (rows and columns)**.
- Schema (data structure) is **predefined**.

### 🔹 Example:

| Student_ID | Name | Department | Marks |
|---|---|---|---|
| 101 | Rahul | CS | 85 |

Each column has a fixed **data type**. You can easily run SQL queries like:

**Real-Life Examples:**

- Banking records
- Airline reservations
- Inventory databases
- Employee details in HR systems

**3. Unstructured Data**

**Definition:**

Unstructured data is raw, unorganized data that does not follow any predefined format or schema.

It can be text, images, audio, video, emails, etc.

## Key Characteristics:

- Does not fit into tables or relational models.
- Difficult to store and analyze using traditional DBMS.
- Usually stored in data lakes or NoSQL systems.
- Requires special tools like Hadoop, Spark, or AI-based text/image processing.

**Example:**

A single folder may contain:

A Word document (Project.docx)
A video file (Lecture.mp4)
An email (email.eml)
A photo (profile.jpg)

A text message ("Meeting postponed to 4 PM")

Each item has valuable data, but it doesn't follow a fixed structure.

- Real-Life Examples:

- Social media posts (Instagram photos, tweets, comments)
- Customer feedback (emails, chat messages)
- CCTV footage
- YouTube videos

# 4. Semi-Structured Data

**Definition:**

Semi-structured data is **partially organized** — it doesn't follow a strict table format like structured data,
but still contains **tags or markers** that make it easier to organize and interpret.
It has a **flexible schema**, meaning data items may not have identical fields,
but there is **some structure** to identify elements.

**Key Characteristics:**

- Stored in **NoSQL databases** like MongoDB, CouchDB.

- Uses **key-value pairs, JSON(JAVASCRIPT OBJECT NOTATION ), XML(EXTENSIBLE MARKUP LANG) , or CSV(COMMA, SEPARATED VALUE) formats**.
- Easier to modify and expand compared to structured data.
- Supports **hierarchical or nested data**

- **Example OF JSON**

```
{
  "StudentID": 101,
  "Name": "Rahul",
  "Subjects": ["DBMS", "OS", "CN"],
  "Marks": {
    "DBMS": 85,
    "OS": 90,
    "CN": 80
  }
}
```

- **EXAMPLE OF XML**

```
<Student>
  <ID>101</ID>
  <Name>Rahul</Name>
  <Department>CS</Department>
  <Marks>85</Marks>
</Student>
```

**You can see it has structure (tags/keys), but not as rigid as a SQL table.**

- **Real-Life Examples:**

- JSON data from APIs
- XML files in web services
- Email headers
- Sensor data from IoT devices

## COMPARISON

| Feature | Structured Data | Semi-Structured Data | Unstructured Data |
|---|---|---|---|
| **Schema** | Fixed & predefined | Flexible | No schema |
| **Storage** | RDBMS (MySQL, Oracle) | NoSQL (MongoDB, Cassandra) | Data lakes, file systems |
| **Format** | Tables (rows & columns) | XML, JSON, CSV | Text, images, videos |
| **Ease of Querying** | Easy (SQL) | Moderate (NoSQL, XPath) | Difficult (needs AI/ML) |
| **Examples** | Student, Employee DB | JSON API, XML files | Emails, videos, social posts |
| **Best Use Case** | Banking, ERP systems | Web data, IoT data | Multimedia, content storage |

- **APPLICATION OF DIFFERENT TYPES OF DATA**

| Scenario | Type of Data | Example |
|---|---|---|
| **Bank transaction record** | Structured | Table with account number, date, amount |
| **Instagram photo caption + image** | Unstructured | Text + image without fixed format |
| **Online shopping order in JSON** | Semi-structured | { "orderID": 101, "items": ["Shoes", "Bag"], "price": 2500 } |

**In Short:**

- **Structured Data** → Like a well-organized Excel sheet.
- **Unstructured Data** → Like a messy folder full of mixed files.
- **Semi-Structured Data** → Like a folder of labeled files (some order, but flexible)

- **ALL ABOUT NoSQL**

## Introduction

## — what is NoSQL?

**NoSQL** OR "Not Only SQL" (or simply "non-relational") databases.
They store and query data using models other than the fixed relational table schema: **key-value, document, column-family, graph, time-series, search indexes**, etc.
NoSQL systems are built for **horizontal scale, flexible schemas.**

**1. Introduction**

| Type | Meaning |
|---|---|
| RDBMS | Relational Database Management System — stores data in tables (rows & columns) using relations. |
| NoSQL | Non-Relational Database System — stores data in different formats (key-value, document, graph, column) instead of tables. |

**2. Data Structure**

| Feature | RDBMS | NoSQL |
|---|---|---|
| Data format | Structured (tables with fixed schema) | Unstructured, Semi-structured, or Structured |
| Schema | Fixed schema (must define columns before inserting data) | Dynamic schema (can add fields anytime) |
| Storage Model | Tables (rows & columns) | Key-value pairs, Documents, Graphs, Columns |

**EXAMPLE :- RDBMS (MySQL, Oracle, PostgreSQL):**

| ID | Name | Age | Department |
|---|---|---|---|
| 1 | Riya | 20 | CSE |

**NoSQL (MongoDB – Document Oriented):**

```
db.students.insertOne({
  "_id": 1,
  "name": "Riya",
  "age": 20,
```

```
  "department": "CSE",
  "skills": ["Java", "Python"]
});
```

**Data stored as a JSON document:**

```
{
 "_id": 1,
 "name": "Riya",
 "age": 20,
 "department": "CSE",
 "skills": ["Java", "Python"]
}
```

- **You can later add "hobbies": "Reading" without changing schema.**

| Feature | RDBMS | NoSQL |
|---|---|---|
| Query language | Structured Query Language (SQL) | Different APIs or query languages (e.g., MongoDB query, Cassandra CQL) |
| Example | SELECT * FROM Student WHERE Age > 18; | db.students.find({ age: { $gt: 18 } }) |

**EXAMPLE OF RDBMS AND NOSQL**

| RDBMS | NoSQL |
|---|---|
| MySQL | MongoDB |
| Oracle | Cassandra |
| PostgreSQL | CouchDB |
| SQL Server | Redis |
| MariaDB | Neo4j (Graph) |

- ## Why do we need NoSQL? (the need — step-by-step)

1. **Web scale & huge data volume** — modern services (social media, ad platforms, e-commerce) must store terabytes/petabytes and serve millions of requests per second.
   *Example:* Netflix stores and serves massive user/viewing data;

   a single RDBMS can become a bottleneck.
2. **Schema flexibility** — product catalogs or user profiles change often; adding a new field in an RDBMS is costly.
   *Example:* An online marketplace wants different attributes per product type (size, material, dimensions).
3. **high throughput reads/writes** — some use cases need microsecond responses and very high write rates.
   *Example:* Session stores, shopping carts
4. **Complex relationships / graph problems** — social networks, recommendations are more naturally modeled as graphs.
   *Example:* Friend-of-friend queries for recommendations.
5. **Geographical distribution & fault tolerance** — global apps need data close to users and survive network partitions.
   *Example:* Multi-region user profile replication.

- **Types of NoSQL databases — explained with real-time examples & snippets**

## 1. Document-Based NoSQL Databases

- **Stores data in documents**, usually in JSON or BSON format.
- Each document is a self-contained unit of data.

- **Example:**
  **MongoDB :- Use Case:**
- Used by **e-commerce websites**, **content management systems**, and **real-time analytics**.
- For example, **Amazon** or **eBay** might store user profiles, order history, and product details as documents.
- **Sample Document:**

```
{
  "userId": "12345",
  "name": "Alice",
  "orders": [
    {"item": "Shoes", "price": 50},
    {"item": "Hat", "price": 20}
  ]
}
```

## 2. Key-Value Stores

- **Data is stored as a pair: key and value.**
- **Super fast and simple — ideal for caching and quick lookups.**

**Example:**

**Redis, Amazon DynamoDB**

**Use Case:**

- **Used in session management, shopping carts, leaderboards in games.**
- **For example, Netflix uses Redis for storing user session data.**
-

**Ex. Key-Value Pair:**

Key: user:1001

Value: {"name": "Bob", "membership": "Premium"}

---

**3. Column-Family Stores (Wide-Column Stores)**

- **Stores data in rows and columns, but columns are grouped into families.**
- **Highly scalable — great for big data applications.**

**Example:**

**Apache Cassandra, HBase :- Use Case:**

- **Used in IoT applications, sensor data, time-series data.**
- **Instagram uses Cassandra to store millions of user photos and comments efficiently.**

**Ex. (in Cassandra):**

**User (Column Family)**

**----------**

**user_id | name | email | age**

**--------|------|---------------|-----**

**1001 | John | john@email.com | 28**

---

## 4. Graph Databases

- **Designed for storing relationships between data (nodes and edges).**
- **Excellent for social networks and recommendation engines.**

**Example:**

**Neo4j, Amazon Neptune**

**Use Case:**

- **Used in Facebook's friend suggestions, LinkedIn recommendations, fraud detection.**
- **For example, Facebook uses graph databases to analyze connections between people.**
-

Example:-

**(Alice) --> [FRIENDS_WITH] --> (Bob)**

## Summary Table:

| Type | Real-Time Example | Use Case Example | Description |
|---|---|---|---|
| **Document Store** | **MongoDB** | **E-commerce product catalogs** | Manages driver and trip information |
| **Key-Value Store** | **Redis, DynamoDB** | **User sessions, caching** | Stores shopping cart and session data |
| **Column Store** | **Apache Cassandra** | **IoT, time-series data, social media logs** | Stores and analyzes streaming data |
| **Graph Database** | **Neo4j, Amazon Neptune** | **Social networks, recommendation engines** | Manages friend connections and recommendations |

## ***Consistency, Transactions & CAP***

- NoSQL systems often use **BASE** (Basically Available, Soft state, Eventual consistency) instead of strict ACID — but many provide **tunable consistency** and limited transactions now (e.g., MongoDB multi-doc transactions, Cassandra lightweight transactions).
- **Choose consistency vs latency vs availability** based on requirements. For many web apps, **eventual consistency** is acceptable (e.g., caches, feeds). For money transfers, strong consistency is required.

# BASE Properties

In traditional **Relational Databases (RDBMS)**, data follows **ACID properties** (**Atomicity, Consistency, Isolation, Durability**) — ensuring **reliability** and **transactional safety**.
**NoSQL databases** are designed for **high scalability**, **performance**, and **availability**, often distributed across multiple servers or regions.

Because of this, **strict ACID rules are difficult to maintain** across distributed systems — especially under the **CAP Theorem** (Consistency, Availability, Partition tolerance).
So, NoSQL systems follow a different model known as **BASE**.

## 2. What BASE Stands For

| Property | Meaning | Description |
|---|---|---|
| **B A–** Basically Available | The system guarantees **availability** even in case of partial failures. | The system always responds — even if some data is temporarily unavailable. |
| **s–** Soft State | The system's state can change **over time**, even without new input. | Data may not be immediately consistent across all nodes. |
| **E** – Eventual Consistency | The system will become **consistent eventually** once all updates are propagated. | After some time, all replicas will hold the same data. |

## 3. Detailed Explanation with Real-Time Examples

### Basically Available

- The system **always gives a response** — success, failure, or a partial result.
- Even if a node fails, other nodes still handle requests.

- **Example:**
  - In **Amazon DynamoDB**, if one server fails, another replica still handles user requests.
    You might see slightly outdated data, but the system remains available — it doesn't crash.

### *Real-time scenario:*

Suppose you're shopping on **Amazon**, and one product's price server is down.
You'll still see a price (maybe from a cached copy).
The system stays **available**, even if the data isn't 100% up-to-date.

---

## Soft State

- The system's state may change **over time** due to replication and background synchronization.
- It doesn't require all nodes to be synchronized instantly.

- **Example:**
  - In **Cassandra**, data written to one node is later replicated to others asynchronously.
  - So, for a short time, different nodes might show slightly different data.

### *Real-time scenario:*

You update your profile picture on **Instagram**.
You see your new photo instantly, but your friend might see the old one for a few seconds.
The state is "soft" — it changes until all replicas sync.

---

## Eventual Consistency

- The system doesn't guarantee **immediate consistency**, but it ensures that **after some time**, all nodes will have the **same data**.

- It's a **trade-off** for high performance and availability.

🔹 **Example:**

- In **MongoDB** or **DynamoDB**, after you update a record, not all users see the new version immediately.
- After replication finishes, **all nodes become consistent**.

## *Real-time scenario:*

When you "like" a post on **Facebook**, it may take a few seconds before the total "likes" count updates for all users.
Eventually, everyone sees the same total — that's **eventual consistency**

1. **BASE vs ACID Comparison**

| **Feature** | **ACID (RDBMS)** | **BASE (NoSQL)** |
|---|---|---|
| Focus | Consistency | Availability and Scalability |
| Consistency | Strict (always consistent) | Eventual (after some delay) |
| State | Stable | Soft (changes over time) |
| Use Case | Banking, transactions | Social media, e-commerce, IoT |
| Example | MySQL, Oracle | MongoDB, Cassandra, DynamoDB |

**In short:**

## BASE = Basically Available, Soft State, Eventually Consistent

## It allows high availability and scalability at the cost of

**immediate consistency — which is ideal for large distributed NoSQL systems.**

## 1. What is Data Consistency Model?

In a **distributed database system** (like NoSQL), data is stored across **multiple servers or nodes**.

A **Data Consistency Model** defines **how and when** changes made to one copy of the data become visible to other users or nodes in the system.

In simple terms:

It decides **how up-to-date and synchronized** your data copies are across the system.

---

**2. Why Consistency Models Matter**

When multiple users or systems **read and write data** at the same time:

- Should all users see the **same data immediately**?
- Or is it okay if some users see **slightly older data** for better speed?

**A. Strong Consistency**

**Definition:**

After a data update, **any read request** (from any node) will return the **most recent value** immediately.

- All users always see the **same, latest data**.
- This ensures correctness but may reduce performance in distributed systems.

**Example:**

- You transfer ₹1000 from your bank account using **an online banking app**.
- Once the transaction completes, both you and the bank system instantly see your new balance.
- There is **no delay** — data is **strongly consistent**.

**Used In:**

- Banking systems
- Stock trading platforms
- Ticket booking systems (to prevent double booking)

**B. Weak Consistency**

**Definition:**

After an update, the system **does not guarantee** that all users will immediately see the latest data.
Eventually, all copies will become consistent after some time.

Weak consistency improves **performance** and **availability**, but at the cost of **immediate accuracy**.

---

**Eventual Consistency**

**Definition:**
After some time, **all replicas will hold the same data**, but **not immediately**.

**Example:**

- You update your **profile picture** on **Instagram**.
- You see the new photo instantly, but your friend might still see your old one for a few seconds.
- After replication completes, everyone sees the same photo.

**Used In:**

- Social media platforms (Instagram, Facebook)

- E-commerce systems (Amazon, Flipkart)

**Example Databases:**

- **Amazon DynamoDB**
- **Cassandra**
- **MongoDB**

# *ACID Vs BASE,*

| Feature | ACID (RDBMS) | BASE (NoSQL) |
|---|---|---|
| **Full Form** | Atomicity, Consistency, Isolation, Durability | Basically Available, Soft State, Eventual Consistency |
| **Focus** | Data integrity and correctness | High availability and scalability |
| **Consistency** | Strict (immediate) | Eventual (after some delay) |
| **Transaction Handling** | Transactions are fully supported | Transactions are often relaxed or partial |
| **System Type** | Centralized, single-node or small clusters | Distributed, multi-node systems |
| **Performance** | Slower for large-scale operations | Faster, handles huge data volumes |
| **Use Case** | Banking, billing, stock trading | Social media, e-commerce, IoT, content delivery |
| **Failure Handling** | Transactions roll back to maintain consistency | System remains available but may show old data temporarily |
| **Example Databases** | MySQL, Oracle, PostgreSQL | MongoDB, Cassandra, DynamoDB |

# ***Comparative Study: RDBMS vs NoSQL***

1. **Data Model**:
    - RDBMS stores data in structured tables with rows and columns.
    - NoSQL uses various formats like key-value pairs, documents (JSON), wide-columns, or graphs.
2. **Schema**:
    - RDBMS uses a fixed schema, requiring predefined table structures.
    - NoSQL has a dynamic schema, allowing flexible and evolving data structures.
3. **Scalability**:
    - RDBMS scales vertically by upgrading server hardware.
    - NoSQL scales horizontally by adding more servers or nodes.
4. **Query Language**:
    - RDBMS uses SQL (Structured Query Language).
    - NoSQL uses non-standard query languages (e.g., MongoDB uses JSON-based queries, Cassandra uses CQL).
5. **Transaction Support**:
    - RDBMS follows ACID (Atomicity, Consistency, Isolation, Durability) for strong consistency.
    - NoSQL follows BASE (Basically Available, Soft state, Eventually consistent) for flexibility and performance.
6. **Use Cases**:
    - RDBMS is ideal for banking systems, HR software, and applications needing strong consistency and relational data.
    - NoSQL is ideal for social media, real-time analytics, IoT, and applications handling large volumes of unstructured or semi-structured data.

7. **Examples**:
    - RDBMS: MySQL, PostgreSQL, Oracle, SQL Server.
    - NoSQL: MongoDB, Cassandra, Redis, Neo4j, Amazon DynamoDB.
8. **Joins and Relationships**:
    - RDBMS supports complex joins and foreign key relationships.
    - NoSQL does not typically support joins; relationships must be handled at the application level or with embedded documents.
9. **Data Integrity**:
    - RDBMS enforces strict data integrity rules through constraints.
    - NoSQL provides more flexibility and sacrifices strict integrity for scalability.
10. **Storage Format**:
    - RDBMS stores structured/tabular data.
    - NoSQL stores data in formats like JSON, BSON, key-value pairs, or graph structures.
11. **Performance with Big Data**:
    - RDBMS can struggle with massive unstructured data sets.
    - NoSQL is optimized for handling high-volume, high-velocity big data.
12. **Flexibility**:
    - RDBMS is rigid; altering schemas can be complex.
    - NoSQL is highly flexible and adapts quickly to data changes.

---

**Real-Life Scenarios**

- Use **RDBMS** for banking, payroll systems, or ERP where consistency and relationships are critical.

- Use **NoSQL** for real-time apps like chat systems, recommendation engines, or social networks needing fast and scalable access to data.

# **MongoDB** (with syntax and usage):

## What is MongoDB?

**MongoDB** is a popular **NoSQL, document-oriented database** used to store and manage large sets of unstructured or semi-structured data. It stores data in flexible, JSON-like **documents**, making it ideal for modern web and mobile applications.

## Key Features of MongoDB

- **No schema needed** (schema-less)
- Stores data in **BSON** (Binary JSON) format
- Supports **embedded documents** and arrays
- High **performance**, **scalability**, and **flexibility**
- Supports **replication** and **sharding** for high availability

**MongoDB Basic Concepts**

| Concept | Description |
|---|---|
| **Database** | A container for collections |
| **Collection** | A group of MongoDB documents (like SQL tables) |
| **Document** | A record in BSON format (like a row in SQL) |
| **Field** | A key-value pair in a document |
| **_id** | A unique identifier automatically assigned |

## MongoDB Syntax (Common Operations)

MongoDB uses JavaScript-like syntax in its shell (or Node.js).

**1. Creating / Switching a Database**

```
use myDatabase
```

---

**2. Creating a Collection**

```
db.createCollection("students")
```

Note: MongoDB creates collections **automatically** when you insert data.

---

**3. Inserting Documents**

**Single Document: ex.**

```
db.students.insertOne({
  name: "Alice",
  age: 22,
  course: "Computer Science"
})
```

**Multiple Documents:**

```
db.students.insertMany([
  { name: "Bob", age: 23, course: "Math" },
  { name: "Charlie", age: 24, course: "Physics" }
])
```

---

**4. Querying Data**

**Get all documents:**

```
db.students.find()
```

**Get with condition:**

```
db.students.find({ age: { $gt: 22 } })
```

***$gt means “greater than”. Other operators: $lt, $eq, $ne, $in***

---

**5. Updating Documents**

**Update one:**

```
db.students.updateOne(
  { name: "Alice" },
  { $set: { age: 23 } }
)
```

**Update many:**

```
db.students.updateMany(
  { course: "Math" },
  { $set: { department: "Mathematics" } }
)
```

---

**6. Deleting Documents**

**Delete one:**

```
db.students.deleteOne({ name: "Bob" })
```

**Delete many:**

```
db.students.deleteMany({ age: { $lt: 23 } })
```

---

**7. Using Indexes**

```
db.students.createIndex({ name: 1 })
```

---

## Real-World Example: E-commerce Product Catalog

```
db.products.insertOne({
  name: "Laptop",
  brand: "Dell",
  price: 750,
  stock: 30,
  specs: {
    ram: "16GB",
    storage: "512GB SSD",
    processor: "Intel i7"
  },
  tags: ["electronics", "computers"]
})
```

# Advantages of MongoDB

- Schema-less (flexible)
- Ideal for JSON-style data
- Easy horizontal scaling (sharding)
- Fast read/write operations
- Great for Agile/DevOps workflows

## When to Use MongoDB

- Dynamic or semi-structured data
- Applications needing rapid development
- Real-time analytics
- Content management systems
- IoT and mobile app backends

# 1. CRUD Operations (Create, Read, Update, Delete)

- **Meaning:**

CRUD are the **four basic database operations** you perform on documents in MongoDB.

| Operation | Command | Description |
|---|---|---|
| **C – Create** | insertOne() / insertMany() | Add new documents |
| **R – Read** | find() | Retrieve documents |
| **U – Update** | updateOne() / updateMany() | Modify existing data |
| **D – Delete** | deleteOne() / deleteMany() | Remove data |

**Real-Time Example — E-commerce Store**

**Database:** shopDB
**Collection:** products

**Create**

```
db.products.insertOne({
  name: "Wireless Mouse",
```

```
  brand: "Logitech",
  price: 899,
  stock: 120
});
// to Adds a new product to the store.
```

---

**Read**

```
db.products.find({ brand: "Logitech" });
```

// to Finds all products from brand “Logitech”.

---

**Update**

```
db.products.updateOne(
  { name: "Wireless Mouse" },
  { $set: { price: 799 } }
);
```

**Delete**

```
db.products.deleteOne({ name: "Wireless Mouse" });
```

// to  Removes the product from the collection.

**1. Indexing**

**What is is indexing?**

**Indexing** is the process of creating a data structure that improves the speed of data retrieval operations.

Without indexing, the database must **scan every document** in a collection (called a **collection scan**) to find matching results.

**Example:**

Suppose you have a users collection with thousands of records.

db.users.find({ email: "user@example.com" }) // Without index = slow

- **Create an Index:**

db.users.createIndex({ email: 1 }) // 1 = ascending order

- **Benefit:**
- Makes searches **faster**
- But requires **more storage** and may slow down **writes**

---

**2. Aggregation**

**What is aggregation:-**

**Aggregation** is used to **process data records** and **return computed results** (like GROUP BY in SQL).

MongoDB uses the **aggregation pipeline**, which passes data through **stages** like $match, $group, $sort, etc.

**Example: Count users by age**

db.users.aggregate([

{ $group: { _id: "$age", count: { $sum: 1 } } }

])

***Benefit:***

- Helps in **data analysis**
- Can perform **complex transformations**

---

**3. MapReduce**

## What is it map reduce

**MapReduce** is a powerful tool for **processing large volumes of data** across distributed systems. It consists of two functions:

- map: Filters and sorts data
- reduce: Aggregates results

It's older and less efficient than the aggregation pipeline in MongoDB but still useful for complex, large-scale processing.

## Example: Word Count in Comments

```
db.comments.mapReduce(
  function () { emit(this.word, 1); },          // map
  function (key, values) { return Array.sum(values); }, // reduce
  { out: "word_count" }
)
```

**Benefit:**

- Useful for **big data analytics**
- Works in **parallel**

---

**4. Replication**

**What is it replication?**

**Replication** is the process of creating **copies of your data** across multiple servers (called a **replica set**).

This ensures:

- **High availability**
- **Fault tolerance**
- **Disaster recovery**

**How it works:**

- **Primary** node handles all writes
- **Secondary** nodes replicate data from the primary
- If primary fails, one secondary is **automatically promoted**

**Example Setup:**

rs.initiate()  // Initialize a replica set

**Benefit:**

- Ensures your app **keeps running** even if a server goes down

---

**5. Sharding**

**What is it sharding?**

**Sharding** is a method of **splitting large data sets** across multiple machines (called **shards**) to improve **read and write performance**.

Used when a **single machine can't handle** the volume of data or traffic.

**Real-Time Example — Online Shopping Site**

MongoDB splits the orders collection based on region:

- **Shard 1:** North India orders
- **Shard 2:** South India orders
- **Shard 3:** West India orders

sh.enableSharding("shopDB")

sh.shardCollection("shopDB.orders", { region: 1 })

Each shard handles fewer records → **faster performance** and **scalable system**.

**Example:**

A collection is sharded by user ID:

sh.enableSharding("myDatabase")

sh.shardCollection("myDatabase.users", { userId: 1 })

Each shard stores a **subset** of the data.

**Benefit:**

- Supports **horizontal scaling**
- Allows MongoDB to handle **huge data volumes**

# Unit 6- Advances in Databases

## PT.1. Emerging Databases :-

refer to new or evolving types **of database systems** that are developed to meet modern data challenges—such as handling big data, unstructured data, real-time analytics, or highly distributed architectures—that traditional databases (like relational databases) struggle with.

**Key Characteristics of Emerging Databases:**

1. Designed for modern needs – such as cloud comp\uting, IoT, AI/ML, and large-scale web applications.
2. Non-traditional models – often move away from the rigid table-based structure of relational databases.
3. Scalable and flexible – support horizontal scaling and can handle a variety of data types.
4. Performance-optimized – for specific workloads like graph processing, time-series data, etc.

**Examples of Emerging Database Types:**

| Type | Description | Example Systems |
|---|---|---|
| NoSQL | Non-relational databases; handle unstructured or semi-structured data. | MongoDB, Cassandra, Couchbase |
| NewSQL | Provide SQL-like capabilities but with better scalability and performance. | CockroachDB, Google Spanner |
| Time-Series DB | Optimized for time-stamped data, such as logs or sensor data. | InfluxDB, TimescaleDB |
| Graph DB | Designed to handle graph-like data structures (nodes and edges). | Neo4j, ArangoDB |
| Document Stores | Store data as JSON-like documents. | MongoDB, CouchDB |

| Type | Description | Example Systems |
|---|---|---|
| Multi-Model DB | Support multiple data models (e.g., key-value + document + graph) in one system. | ArangoDB, OrientDB |
| Cloud-Native DB | Built specifically for cloud environments with elastic scaling. | Amazon Aurora, Firebase Realtime DB |
| Blockchain DB | Immutable, decentralized databases with distributed ledger technology. | BigchainDB, Hyperledger Fabric |

**Why Are They "Emerging"?**

Because they:

- Address **gaps left by traditional relational databases**.
- Are relatively **new in adoption** and still evolving.
- Are being used in **cutting-edge applications**, like AI, IoT, and real-time analytics.

## PT.1.i Active Databases

- **Definition:**

An **Active Database** is a database system that automatically **responds to certain events or conditions** by executing pre-defined rules. These rules are called **triggers**.

- **Key Concept: ECA Rules**

Active databases use **ECA rules**:

- **E**vent – What triggers the rule (e.g., data update)
- **C**ondition – A test to check whether the rule should execute
- **A**ction – What the system should do automatically

**Example:**

prob.state.:- If a customer's balance goes below $100, automatically send a warning email.

Solution:- CREATE TRIGGER check_balance

AFTER UPDATE ON accounts

FOR EACH ROW

WHEN (NEW.balance < 100)

BEGIN

-- Action: Send notification

END;

- **Use Cases:**
  - Real-time monitoring
  - Automated notifications
  - Enforcing business rules
  - Intrusion**( Unauthorized Access or Attack)** detection

---

## PT.1.ii. Deductive Databases

- **Definition:**

A **Deductive Database** is a database system that can make **logical inferences** (***Logical inference*** *is the process of* ***deriving new information or conclusions*** *from known facts*) based on rules and facts stored in the database. It's based on **logic programming**, especially **Datalog**, a subset of Prolog.

- **How it works:**

- Stores **facts** (data)
- Stores **rules** (logic)
- Uses a **query engine** to deduce new information not explicitly stored

**Example:**

father(john, mike).

father(mike, tom).

**Rules (logical inference rules):**

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

This rule says:

X is a **grandparent** of Y **if** X is a parent of Z **and** Z is a parent of Y.

**Logic of finding relation**

**Inference:**

- john is a parent of mary
- mary is a parent of lisa
- Therefore, john is a **grandparent** of lisa

ancestor(X, Y) :- father(X, Y).

ancestor(X, Y) :- father(X, Z), ancestor(Z, Y).

Querying ancestor(john, tom). would return true, even though it's not explicitly stored—it's **inferred**.

- **Use Cases:**
  - Expert systems
  - AI reasoning tasks
  - Complex rule-based querying

- Semantic web and knowledge graphs

---

**Comparison Table:**

| Feature | Active Database | Deductive Database |
|---|---|---|
| Focus | Automation & reactions | Logical inference & reasoning |
| Trigger Type | Event-based (ECA) | Rule-based (logic programming) |
| Query Language | SQL with triggers | Datalog, Prolog |
| Example Use | Auto alerts, monitoring | Expert systems, AI reasoning |
| Key Feature | Immediate action on data events | Derive new facts from existing data |

## PT.1.iii. Main Memory Databases (MMDBs)

- **Definition:**

A **Main Memory Database** (also called an **In-Memory Database**) stores **all data in the main RAM** (not on disk), enabling extremely **fast data access** compared to traditional disk-based databases.

---

- **Key Features:**
  - Very **high performance** (low latency: ***Low latency*** *refers to a* ***short delay*** *between a user's action and the system's response*.)
  - Data is **volatile (***The data is temporary and will be lost when the system is turned off, restarted, or crashes.)* unless backed up or persisted
  - Often used in **real-time systems** and **high-frequency applications**

---

**Example:**

Let’s say a stock trading platform must process thousands of transactions per second.

Speed is critical. Instead of reading and writing to disk, it stores all trading data **in memory**.

**Example System:**

- **Redis** – Key-value in-memory store
- **SAP HANA** – In-memory relational database
- **VoltDB** – Designed for real-time analytics

**Redis Example:**

SET user:101:name "Alice"

GET user:101:name

Here, data is stored and accessed directly from RAM – almost instantly.

---

- **Use Cases:**
  - Real-time analytics (e.g., fraud detection)
  - Gaming leaderboards
  - Caching layers
  - High-frequency trading systems

---

## PT.1.iv:- Semantic Databases

- **Definition:**

**Understands the meaning (semantics) of data** and how different pieces of data are **related** to each other.

Instead of just storing raw data like a traditional database, a semantic database stores:

- **Data + Meaning**
- **Data + Relationships**
- **Data + Context**

---

🔹 **Simple Example:**

**Traditional Database:**

Table: Books

| ID | Title | Author |

| 1 | Hamlet | William Shakespeare |

The system sees this as just rows and columns.

---

**Semantic Database:**

Stores:

- Hamlet is a **Book**
- William Shakespeare is a **Person**
- William Shakespeare is the **Author of Hamlet**

It understands the **relationship** between them and what they **mean** — not just how they're written.

---

🔹 **Key Technologies:**

- RDF (Resource Description Framework)
- OWL (Web Ontology Language)
- SPARQL (Query language)

---

**Example:**

*Let's say we store this fact:*

*"Albert Einstein was born in Germany."*

*This is stored as a triple:*

- ***Subject:*** *Albert_Einstein*
- ***Predicate:*** *was_born_in*
- ***Object:*** *Germany*

***SPARQL Query:***

```
SELECT ?person WHERE {
  ?person <was_born_in> <Germany> .
}
```

*This will return **Albert Einstein**, and any other person born in Germany.*

---

**Example Systems:**

- **Apache Jena** – Java framework for building Semantic Web apps
- **Virtuoso** – RDF store and SPARQL endpoint
- **Stardog**, **GraphDB**, **Blazegraph**

---

- **Use Cases:**
  - Knowledge graphs (e.g., Google Knowledge Graph)
  - Biomedical data (e.g., linking diseases to genes)
  - Semantic search engines
  - Linked Data and the Semantic Web

---

**Comparison Table:**

| Feature | Main Memory Database | Semantic Database |
|---|---|---|
| Storage | In RAM | Usually disk-based with semantic structure |
| Focus | Speed and performance | Meaning and relationships |
| Data Model | Tables / Key-Value | RDF triples (subject-predicate-object) |
| Query Language | SQL, NoSQL | SPARQL |
| Best For | Real-time processing | Complex relationship queries |
| Examples | Redis, SAP HANA | Apache Jena, Stardog |

## PT.2. What are Complex Data Types?

In Database Management Systems (DBMS), Complex Data Types refer to non-primitive or structured data types that store more complicated or multi-part data than simple types like INTEGER, VARCHAR, or DATE.

---

In Simple Words:

Complex Data Types are used to store multiple values, nested data, or structured objects inside a single column.

---

- Examples of Complex Data Types:

| Type | Description | Example |
|---|---|---|
| Array | Stores a list of values in one column | {90, 85, 88} (student marks) |
| Record/Struct | A group of fields combined into one type | (name: 'Ravi', age: 21) |
| Nested Table | A table within a table | A column that stores a list of products in an order |
| JSON/XML | Hierarchical data stored in a single field | JSON object with multiple nested fields |
| User-Defined Types (UDTs) | Custom types combining multiple attributes | Address(city TEXT, zip TEXT) |

**Complex Data Types** are data types that can **store multiple values** or **structured data** within a single field or variable. Unlike **primitive types** (like int, char, float), complex types can store **collections**, **hierarchies**, or **composite structures** such as arrays, records, JSON objects, multimedia, etc.

---

**Types of Complex Data Types (with Examples):**

| **Complex Data Type** | **Description** | **Example** |
|---|---|---|
| **Array** | A collection of elements of the same type | int[] scores = {85, 90, 75} |
| **Record / Struct** | A group of related data items (fields) | {name: "Alice", age: 25, city: "Delhi"} |
| **Set / List** | An unordered (set) or ordered (list) group of values | ["apple", "banana", "mango"] |

| Complex Data Type | Description | Example |
|---|---|---|
| **JSON / XML** | Semi-structured hierarchical data | {"user": {"id": 1, "name": "John"}} |
| **Multimedia (BLOBs)** | Images, audio, video, stored in binary format | Profile photo stored as a BLOB |
| **Spatial Data** | Geographic data like coordinates or shapes | POINT(40.7128, -74.0060) |
| **Temporal Data** | Complex time-based data | timestamp with timezone |

---

- **1. Array Example (SQL):**

```
CREATE TABLE Students (
   id INT,
   name VARCHAR(50),
   marks INT[]
);
```

Here, a single column marks can hold multiple integer values like {85, 90, 78}.

---

- **2. JSON Example (NoSQL / SQL with JSON support):**

```
CREATE TABLE Employees (
   id INT,
   data JSON
);
```

Data in data column could be:

```
{
```

```
  "name": "Alice",
  "skills": ["Python", "SQL", "ML"],
  "address": {"city": "Mumbai", "zip": "400001"}
}
```

---

- **3. Multimedia (BLOB) Example:**

```
CREATE TABLE Images (
    id INT,
    name VARCHAR(100),
    photo BLOB
);
```

Stores binary data for image files, useful for apps handling photos or videos.

---

- **4. Spatial Data Example (PostGIS - extension of PostgreSQL):**

```
CREATE TABLE Locations (
    id SERIAL,
    name TEXT,
    position GEOGRAPHY(POINT, 4326)
);
```

This allows storage and querying of geographic coordinates.

---

- **5. Record / Struct in PL/SQL (Oracle):**

```
TYPE Student IS RECORD (
    id INT,
    name VARCHAR(50),
```

```
    gpa NUMBER
);
```

Used to group multiple fields under one logical unit.

---

**Why Are Complex Data Types Useful?**

- Handle **real-world, diverse data** (e.g., multimedia, user profiles)
- Support **modern applications** like GIS, AI, multimedia apps
- Allow for **more expressive data modeling**
- Essential for **NoSQL and semi-structured data** systems

---

**In Short:**

| **Simple Data Types** | **Complex Data Types** |
|---|---|
| Single values (e.g., INT, CHAR) | Composite or structured values |
| Easy to query and store | Require specialized handling |
| Flat structure | Hierarchical or nested |

**Use Cases:**

- E-commerce (e.g., storing order items as arrays)
- Location data (e.g., nested latitude/longitude)
- User profiles (e.g., storing settings in JSON)
- IoT/sensor data (complex readings in one record)

## PT2.i:- What is Semi-Structured Data?

- Definition:

Semi-structured data is a type of data that does not conform to a fixed schema (like in relational databases) but still has some organizational properties, such as tags or markers to separate data elements.

It's more flexible than structured data and more organized than unstructured data.

---

- Key Characteristics:

Does not use strict tables and columns

Often stored in text-based formats (e.g., JSON, XML, YAML)

Can represent nested and hierarchical information

Schema can vary across records or change over time

**What Are Semi-Structured Data Models?**

A Semi-Structured Data Model is a way of organizing and storing data that:

- Doesn't follow a strict fixed schema,
- But still uses tags or markers to show relationships between data elements.

# PT.3, Nested Data Types

**PT.3.** JSON, **PT.3ii** XML.

**What Are Nested Data Types?**
**Nested data types** refer to data structures that contain **other data structures** inside them—like objects inside objects or arrays inside objects.
This is common in **semi-structured data formats** such as **JSON** and **XML**, where data is often **hierarchical** or **multi-level**.

---

- **1. Nested Data Types in JSON**

**what is JSON?**

- **JSON (JavaScript Object Notation)** is a lightweight format used to represent **structured and nested data**.
- It uses **key-value pairs** and supports **objects**, **arrays**, **strings**, **numbers**, etc.

## JSON Example with Nesting:

```
{
 "name": "Ravi",
 "age": 28,
 "address": {
  "street": "123 MG Road",
  "city": "Mumbai",
  "zip": "400001"
 },
 "skills": ["Python", "Java", "SQL"],
 "education": [
  {
   "degree": "B.Sc",
   "year": 2016
  },
  {
   "degree": "M.Sc",
   "year": 2018
  }
 ]
}
```

## Nested Elements Explained:

- "address" is a **nested object** inside the main object.
- "skills" is a **nested array** of strings.
- "education" is a **nested array of objects**, each containing degree and year.

---

## ◇ 2. Nested Data Types in XML

**What is XML?**

- **XML (eXtensible Markup Language)** is a tag-based format used to represent **hierarchical data**.
- Tags are nested inside one another to show relationships.

## XML Example with Nesting:

```
<employee>
  <name>Priya</name>
  <age>30</age>
  <address>
    <street>456 Park Street</street>
    <city>Delhi</city>
    <zip>110001</zip>
  </address>
  <skills>
    <skill>Python</skill>
    <skill>Data Analysis</skill>
    <skill>SQL</skill>
  </skills>
  <education>
    <degree>
      <type>B.Tech</type>
      <year>2015</year>
    </degree>
    <degree>
      <type>MBA</type>
      <year>2018</year>
    </degree>
  </education>
</employee>
```

## Nested Elements Explained:

- <address> is a **nested tag** inside <employee>.
- <skills> contains multiple <skill> elements → **array-like structure**.
- <education> contains multiple <degree> elements, each with its own nested data.

---

# Key Differences between JSON and XML Nesting

| Feature | JSON | XML |
|---|---|---|
| Structure | Object-based (key-value pairs) | Tag-based (open/close tags) |
| Syntax | Uses {}, [], and : | Uses <tag>...</tag> |
| Readability | Concise and cleaner for developers | Verbose but widely used in enterprise |

| Feature | JSON | XML |
|---|---|---|
| Array Support | Native arrays (e.g., ["a", "b"]) | Uses repeated elements (e.g., <item>) |
| Use in Programming | Easy to parse in JavaScript, Python | Common in data exchange (SOAP, RSS) |

---

# Summary:

- **JSON and XML** both support **nested data types** to represent real-world complex data.
- Nesting allows you to represent data like **addresses, lists, profiles, and objects** within other objects.
- These formats are heavily used in **APIs**, **NoSQL databases**, **web services**, and **data exchange**.

# Summary table:

| Feature | Description | Example Format |
|---|---|---|
| Flexible Schema | No fixed structure | JSON, XML |
| Hierarchical | Nested data | JSON arrays, XML tags |
| Incomplete Data | Fields can be missing | JSON without all keys |
| Self-Describing | Keys or tags define data | "name": "Ali" |
| Human/Machine Readable | Easy for both to read | JSON, YAML |
| Data Exchange Friendly | Used in APIs and services | REST API JSON response |

---

🟩 Examples of Semi-Structured Data:

---

## *1. JSON (JavaScript Object Notation):*

***A popular format for APIs and NoSQL databases.***

{

```
  "name": "Alice",
  "age": 30,
  "skills": ["Python", "SQL"],
  "address": {
    "city": "Mumbai",
    "zip": "400001"
  }
}
```

Key-value pairs

Supports arrays and nested objects

Common in NoSQL databases like MongoDB

---

## 2. XML (eXtensible Markup Language):

***Used in web services, document storage.***

```
<employee>
  <name>Alice</name>
  <age>30</age>
  <skills>
    <skill>Python</skill>
    <skill>SQL</skill>
  </skills>
</employee>
```

Tag-based

Hierarchical

More verbose than JSON

---

## 4. Email:

**An email is semi-structured:**

Structured parts: Subject, From, To, Date

Unstructured part: Body content

---

## 5. HTML:

**Web pages are often semi-structured:**

```
<div>
  <h1>Welcome</h1>
  <p>This is a message.</p>
</div>
```

***Uses tags to structure content***

***Not strictly a database format but commonly used in web data***

---

## Comparison Table:

| Feature | Structured Data | Semi-Structured Data | Unstructured Data |
|---|---|---|---|
| Format | Fixed schema (tables) | Flexible schema (tags/keys) | No schema |
| Example | SQL tables | JSON, XML, YAML | Videos, images, plain text |
| Storage | RDBMS (e.g., MySQL) | NoSQL DBs (e.g., MongoDB) | File systems, media |

| Feature | Structured Data | Semi-Structured Data | Unstructured Data |
| --- | --- | --- | --- |
| Schema changes | Hard | Easy | Not applicable |

---

## ✅ ***Use Cases of Semi-Structured Data:***

Web APIs (returning JSON or XML)

NoSQL databases like MongoDB, CouchDB

Configuration files (DevOps and cloud services)

Log files, sensor data

Data exchange between systems

---

## **Real-World Example:**

In MongoDB, you might store customer profiles like:

```
{
 "name": "Ravi",
 "email": "ravi@example.com",
 "orders": [
  {"id": 101, "product": "Laptop"},
  {"id": 102, "product": "Mouse"}
 ]
}
```

No predefined schema is required — each customer could have different fields or structure.

---

## Summary Table:

# PT4 Object Orientation: PT4.iObject-Relational Database System,

**Object Orientation (OO)**

- **Definition:**

**Object Orientation** is a **programming and design approach** that models data as **"objects"**, which combine **data (attributes)** and **behavior (methods)** into one unit.

## Object-Relational Database Systems (ORDBMS)

- **Definition:**

An **Object-Relational Database Management System** combines features of **Relational Databases (RDBMS)** and **Object-Oriented Databases (OODBMS)**.
It extends the traditional relational model to support **complex data types, inheritance, and user-defined objects** — while still using **SQL** as its query language.

---

- **In Short:**

ORDBMS = Relational DB + Object-Oriented Features

---

- **Key Features of ORDBMS:**

| Feature | Description | Example |
|---|---|---|
| **User-Defined Types (UDTs)** | Create custom data types | CREATE TYPE Address AS (city TEXT, zip TEXT); |
| **Table Inheritance** | Tables can inherit structure from other tables | Student inherits from Person |
| **Complex/Nested Types** | Fields can be records or arrays | marks INTEGER[] or address Address |
| **Object Identity** | Each row can behave like an object | Rows can reference each other like objects |

---

**Example in PostgreSQL (an ORDBMS):**

```
-- Create a complex type
```

```
CREATE TYPE Address AS (
  city TEXT,
  zip TEXT
);

-- Use the type in a table
CREATE TABLE Customer (
  id INT,
  name TEXT,
  home_address Address
);

-- Insert data with a nested structure
INSERT INTO Customer VALUES (1, 'Ravi', ('Mumbai', '400001'));
```

Here:

- Address is a **user-defined object type**
- It's used as a column in a relational table — mixing relational and object-oriented features

---

◇ **Examples of ORDBMS Software:**

- **PostgreSQL**
- **Oracle Database (with object extensions)**
- **IBM Db2**
- **Informix**

---

**Benefits of ORDBMS:**

- Better modeling of **real-world data**
- Supports **complex types** and **relationships**
- Keeps **SQL compatibility**
- Ideal for apps needing both relational and object features (e.g., GIS, CAD, financial apps)

---

**Difference Between RDBMS and ORDBMS:**

| **Feature** | **RDBMS** | **ORDBMS** |
|---|---|---|
| Data Types | Simple (int, varchar) | Simple + complex/user-defined types |
| Schema | Flat tables | Tables + object structures |
| Inheritance | Not supported | Supported |
| Example | MySQL, SQLite | PostgreSQL, Oracle (with objects) |

**PT4.ii.**Table Inheritance, **PT4.iii** Object-Relational Mapping.

# Table Inheritance (in ORDBMS)

◇ **Definition:**

**Table Inheritance** is a feature in **Object-Relational Databases** (like PostgreSQL) that allows a table to **inherit columns and constraints from a parent table** — just like **class inheritance** in object-oriented programming. It helps model **real-world hierarchical relationships** between entities.

---

**Example:**

Suppose we have a general table Person and two specialized types: Student and Teacher.

☑ **Step 1: Create the parent table**

```
CREATE TABLE Person (
  id SERIAL PRIMARY KEY,
  name TEXT,
  age INT
);
```

☑ **Step 2: Create child tables using INHERITS**

```
CREATE TABLE Student (
  course TEXT,
  grade CHAR(1)
) INHERITS (Person);

CREATE TABLE Teacher (
  subject TEXT,
  salary NUMERIC
) INHERITS (Person);
```

Now:

- Student has columns: id, name, age, course, grade
- Teacher has columns: id, name, age, subject, salary

☑ **Step 3: Insert and query**

```
INSERT INTO Student (name, age, course, grade)
VALUES ('Asha', 20, 'Math', 'A');

SELECT * FROM Person;
```

By default, PostgreSQL **does not return rows from child tables** when querying the parent. Use:

```
SELECT * FROM ONLY Person;  -- Parent only
SELECT * FROM Person;       -- Includes child rows
```

---

☑ **Benefits of Table Inheritance:**

- Models **hierarchies** (e.g., employees → managers, interns)
- Promotes **code/data reuse**
- Clean mapping between **object classes and tables**

---

## ◇ 2. Object-Relational Mapping (ORM)

◇ **Definition:**

**Object-Relational Mapping (ORM)** is a programming technique used to **map database tables to programming language objects (classes)**.

ORM frameworks let developers:

- Work with database data **as objects**
- Avoid writing raw SQL
- Automatically handle **CRUD operations**

---

**Example in Python using SQLAlchemy (an ORM library)**

**Define class mapped to table:**

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Student(Base):
    __tablename__ = 'students'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    course = Column(String)
    grade = Column(String)
```

**ORM Benefits:**

- Student is both a **Python class** and represents a **DB table**.
- You can do:

```
new_student = Student(name="Rahul", course="Science", grade="A")
session.add(new_student)
session.commit()
```

No need to write:

```
INSERT INTO students (name, course, grade) VALUES ('Rahul', 'Science', 'A');
```

**Popular ORM Frameworks:**

| Language | ORM Tool |
|---|---|
| Python | SQLAlchemy, Django ORM |
| Java | Hibernate |
| C# | Entity Framework |
| PHP | Doctrine, Eloquent (Laravel) |

---

# Relationship Between Table Inheritance and ORM:

- **Table Inheritance** is a **database feature** that supports object-oriented modeling.
- **ORM** is a **programming tool** that helps map **object-oriented classes** to **database tables**, often supporting inheritance too.

---

# Summary Table

| Concept | Description | Example |
|---|---|---|
| **Table Inheritance** | One table inherits columns from another in DB | CREATE TABLE Student INHERITS (Person); |
| **ORM** | Programming objects mapped to DB tables | class Student(Base): __tablename__ = 'students' |
| **Used In** | ORDBMS like PostgreSQL | Languages like Python, Java, etc. |
| **Goal** | Reuse schema, model hierarchies | Access DB using OOP code |

# PT5. Spatial Data:

**What is Spatial Data?**

Spatial data (also called geospatial data or geographic data) is information that identifies the geographic location of features and boundaries on Earth.

It describes where things are in space, using coordinates like latitude and longitude or other spatial references.

---

◇ Types of Spatial Data:

1. Vector Data
   Represents features as discrete objects:
   - Points (e.g., a city location)
   - Lines (e.g., roads, rivers)
   - Polygons (e.g., country borders, lakes)
   -
2. Raster Data
   Represents features as a grid of cells or pixels:
   - Satellite images
   - Aerial photographs
   - Digital Elevation Models (DEM)

---

◇ Examples of Spatial Data:

| Type | Example | Description |
|---|---|---|
| Point | GPS location of a restaurant | A single coordinate pair (lat, long) |
| Line | A road or river | Connected points forming a path |
| Polygon | City boundaries, lakes | Areas defined by closed lines |
| Raster | Satellite image | Grid of pixels with color or values |

---

◇ Uses of Spatial Data:

- Mapping and Navigation (Google Maps, GPS devices)
- Urban Planning (zoning, infrastructure)
- Environmental Monitoring (deforestation, weather patterns)
- Disaster Management (flood zones, earthquake risk areas)
- Location-Based Services (finding nearby restaurants or gas stations)

---

◇ How is Spatial Data Stored?

Specialized databases called Spatial Databases (e.g., PostGIS extension for PostgreSQL, Oracle Spatial) support storing, indexing, and querying spatial data using standards like GIS (Geographic Information Systems).

---

◇ Simple Example of Spatial Data (Vector):

A point representing a city in GeoJSON (a common spatial data format):

```
{
  "type": "Point",
  "coordinates": [77.2090, 28.6139]  // Longitude, Latitude of New Delhi
}
```

---

Summary:

- Spatial data represents locations and shapes on the Earth's surface.

- Can be points, lines, polygons, or rasters.
- Essential for mapping, GIS, and location-aware applications.

**PT5.i.**
Geographic Data, **PT5.ii.**Geometric Data.

## Geographic Data

### 🔹 What is Geographic Data?

- **Geographic Data** refers to spatial information that is tied to **locations on the Earth's surface**.
- It includes data with a **geodetic coordinate system** like latitude and longitude, which takes the Earth's curvature into account.
- Geographic data uses a **geographic coordinate system** (e.g., WGS 84 used in GPS).

### 🔹 Key Features:

- Coordinates represent positions **on the globe**, taking into account Earth's shape (ellipsoid/spheroid).
- Used for applications involving real-world earth locations (maps, GPS, GIS).

### 🔹 Example:

A point representing New York City using latitude and longitude:

Latitude: 40.7128° N

Longitude: 74.0060° W

In GeoJSON format:

```
{
  "type": "Point",
  "coordinates": [-74.0060, 40.7128]  // Longitude, Latitude
}
```

---

## 2️⃣ Geometric Data

### 🔹 What is Geometric Data?

- **Geometric Data** is spatial data defined in a **flat, Cartesian coordinate system** (2D or 3D).
- It treats space as flat — it does **not consider Earth's curvature**.
- Commonly used in CAD, computer graphics, and local maps.

🔹 **Key Features:**

- Coordinates are usually simple (x, y) or (x, y, z) values.
- Ideal for representing shapes, drawings, and local spatial relationships.
- Easier for mathematical calculations like distance and area on a flat plane.

🔹 **Example:**

A polygon representing a park in a city map using Cartesian coordinates:

POLYGON((10 20, 30 40, 50 60, 10 20))

Here, numbers represent x and y coordinates on a flat grid.

---

🔹 **Difference Between Geographic and Geometric Data**

| Aspect | Geographic Data | Geometric Data |
|---|---|---|
| Coordinate System | Earth-centered, spherical (lat, long) | Flat, Cartesian (x, y, z) |
| Measurement | Great-circle distances, geodetic calculations | Euclidean distances, simple math |
| Use Case | Global positioning, maps, GIS applications | CAD, local maps, spatial analysis |

| Aspect | Geographic Data | Geometric Data |
| --- | --- | --- |
| Complexity | More complex due to Earth's curvature | Simpler calculations, flat surface |

---

- **Use in DBMS:**
  - Many spatially enabled DBMS like **PostGIS (PostgreSQL)** support both data types.
  - **Geographic data types** are stored using SRIDs (Spatial Reference System Identifiers) to define the coordinate system.
  - **Geometric data types** usually assume a planar coordinate system.