**MongoDb**

1-**NoSQL**

a type of database that doesn’t use traditional tables like SQL databases. It’s designed to handle unstructured, semi-structured, or very large data.

 **Key Features of NoSQL:**

* Non-Relational
* Schema-less
* Horizontal Scaling
* Flexible Data Types
* High Availability and Fault Tolerance

 **Types of NoSQL Databases:**

* **Document-Oriented** (e.g., MongoDB, CouchDB)
* **Key-Value Stores** (e.g., Redis, DynamoDB)
* **Column-Family Stores** (e.g., Cassandra, HBase)
* **Graph Databases** (e.g., Neo4j, ArangoDB)

 **When to Use NoSQL:**

* When dealing with large or changing data.
* When you need to scale your database horizontally.
* For real-time applications.

**Why NoSQL (like MongoDB) was Developed**

* Traditional SQL databases struggled with huge amounts of unstructured or rapidly changing data.
* Applications needed to scale horizontally (across many servers) easily.
* Developers needed faster, more flexible ways to build and adapt applications.

**Is NoSQL Better for Big Projects?**

* It depends on the type of project.
* **NoSQL** is better for projects with big, fast-changing, unstructured data (e.g., social media apps, big data, real-time apps).
* **SQL** is better when strict consistency, structured data, and complex transactions are needed (e.g., banking apps).

**Is NoSQL Better for Big Projects?**

* It depends on the type of project.
* **NoSQL** is better for projects with big, fast-changing, unstructured data (e.g., social media apps, big data, real-time apps).
* **SQL** is better when strict consistency, structured data, and complex transactions are needed (e.g., banking apps).

2-**What is MongoDB?**

MongoDB is a NoSQL database. It stores data in a flexible, JSON-like format called BSON instead of traditional tables.

* **Document-Oriented**: Data is stored in documents (similar to JSON objects).
* **Schema-less**: Documents in the same collection can have different fields.
* **High Scalability**: Handles large amounts of data and traffic easily.

3-**Difference Between MongoDB and SQL Server**

* **SQL Server**:
  + Relational Database (RDBMS).
  + Stores data in tables with rows and columns.
  + Uses SQL queries to interact with data.
  + Requires a fixed schema (structure).
* **MongoDB**:
  + NoSQL Database.
  + Stores data in collections and documents.
  + Flexible schema, no strict structure.
  + Data is retrieved using a query language similar to JSON.

**When to Use Each**

* **Use SQL Server**:
  + When data is highly structured.
  + When you need complex queries, transactions, and relationships (joins).
  + For financial, banking, or enterprise applications.
* **Use MongoDB**:
  + When data is semi-structured or changes frequently.
  + When you need fast development and flexibility.
  + For big data applications, real-time analytics, IoT, and content management.

**Which is Faster?**

* **MongoDB** can be faster when dealing with large volumes of unstructured data and quick reads/writes.
* **SQL Server** can be faster for complex transactions, heavy relationships, and consistent structured data.

**Why SQL Server is Mostly Used**

* Mature and well-known technology.
* Strong support for transactions, consistency, and complex queries.
* Enterprise-grade tools and reporting.
* Many developers are trained on relational databases.

**Why NoSQL (like MongoDB) was Developed**

* Traditional SQL databases struggled with huge amounts of unstructured or rapidly changing data.
* Applications needed to scale horizontally (across many servers) easily.
* Developers needed faster, more flexible ways to build and adapt applications.

4- **Basic Concepts**

* **Database** → Like a folder (group of collections).
* **Collection** → Like a table (group of documents).
* **Document** → Like a row (but JSON format).

5- Operations

**createCollection**: Create a new collection.

**dropCollection**: Drop a collection from the database.

1. **Create Operations:**

* **insertOne**: Insert a single document into a collection.
* **insertMany**: Insert multiple documents into a collection.

### Read Operations:

### find: Retrieve multiple documents from a collection based on a query.

* **findOne**: Retrieve a single document that matches the query.

### 3. Update Operations:

* **updateOne**: Update a single document that matches the query.
* **updateMany**: Update multiple documents that match the query.
* **replaceOne**: Replace an existing document with a new one (entire document gets replaced).
* **deleteOne**: Delete a single document that matches the query.
* **deleteMany**: Delete multiple documents that match the query.

6- Aggregation

Aggregation is used to process and analyze data in MongoDB, like filtering, grouping, or transforming data.

* **aggregate**: Perform complex queries using aggregation pipelines.

db.collection('users').aggregate([

{ $match: { age: { $gt: 25 } } },

{ $group: { \_id: "$age", count: { $sum: 1 } } }]);

* **$group**: Group data by a field.

db.collection('users').aggregate([

{ $group: { \_id: "$age", total: { $sum: 1 } } }

]);

**JOIN:**

Syntax of $lookup (JOIN in MongoDB):

db.orders.aggregate([{

$lookup: {

from: "products", // The collection to join (like the second table)

localField: "productId", // Field in the orders collection

foreignField: "\_id", // Field in the products collection

as: "productDetails" // The name of the array field to add to each order

} }

**$match**: Filter the data for the aggregation pipeline.

db.collection('users').aggregate([

{ $match: { age: { $gt: 25 } } }]);

**$project**: Include or exclude fields in the aggregation result.

db.collection('users').aggregate([{ $project: { name: 1, age: 1 } }]);

**$sort**: Sort documents in the aggregation pipeline.

db.collection('users').aggregate([{ $sort: { age: -1 } }]);

7- Querying

* **$eq**: Equal to a value db.collection('users').find({ age: { $eq: 30 } });
* **$gt / $lt**: Greater than / Less than db.collection('users').find({ age: { $gt: 25 } });
* **$in**: Match any of the values in the array db.collection('users').find({ age: { $in: [25, 30] } });
* **$and / $or**: Combine multiple conditions db.collection('users').find({ $and: [{ age: { $gt: 25 } }, { status: "active" }] });

### ****8. Indexing:****

Indexes help improve query performance by allowing MongoDB to find documents more efficiently.

* **createIndex**: Create an index on a field to speed up queries.
* **dropIndex**: Drop an index.

# Data Modeling in MongoDB

**Data Modeling** means **how you organize and design your data** in the database.

It’s **how you decide**:

* What collections to create
* What fields each document should have
* How documents are related to each other

Good modeling makes your app **faster**, **simpler**, and **easier** to maintain!

## 🔵 MongoDB Data Structure

In MongoDB:

* Data is stored in **collections** (like tables in SQL).
* Each collection holds **documents** (like rows in SQL).
* Documents are **JSON-like** (they look like { key: value }).

# Two Main Ways to Model Data

## 1. **Embedding** (Denormalization)

You put **related data inside the same document**.

✅ Fast reads  
✅ Fewer queries  
✅ Good for data that always comes together.

**Example:**

{ "name": "John", "email": "john@example.com","addresses": [ { "city": "New York", "street": "5th Avenue" }, { "city": "Los Angeles", "street": "Sunset Blvd" }]}

## 2. **Referencing** (Normalization)

You **split related data into different collections** and link them by an ID.

Less duplication  
Good when data is large or shared between many documents.

{ "name": "John", "email": "john@example.com", "addressIds": [ObjectId("123"), ObjectId("456")]}

{ "\_id": ObjectId("123"), "city": "New York", "street": "5th Avenue" }

{ "\_id": ObjectId("456"), "city": "Los Angeles", "street": "Sunset Blvd" }

When you **reference documents** using **IDs**, you usually need to **join** or **combine** data across multiple collections. In MongoDB, we don’t have **joins** like in SQL, but we can use **lookup operations** to achieve similar results.

**Example: Using $lookup for "Join"-like Operations**

You can use the **$lookup** aggregation stage to join the referenced document with the main document.

db.orders.aggregate([

{ $lookup: {

from: "users", // The collection we are joining with

localField: "userId", // Field in orders collection (reference)

foreignField: "\_id", // Field in users collection (the referenced ID)

as: "userDetails" // The result will be stored in this field}}]);

**Transaction**

A **Transaction** means:

A group of database operations that must **all succeed together** or **all fail together**.

If everything works → changes are saved.  
 If anything fails → everything is undone.

using var session = await \_mongoClient.StartSessionAsync();

session.StartTransaction();

try

{

// First operation: Deduct from User A

await usersCollection.UpdateOneAsync(

session,

u => u.Id == "UserA",

Builders<User>.Update.Inc(u => u.Balance, -100)

);

// Second operation: Add to User B

await usersCollection.UpdateOneAsync(

session,

u => u.Id == "UserB",

Builders<User>.Update.Inc(u => u.Balance, 100) );

// Commit if all succeed

await session.CommitTransactionAsync();

}

Catch{// Rollback if anything fails await session.AbortTransactionAsync();}

**Enforcing Rules in MongoDB:**

MongoDB is **flexible** in that it allows you to store documents with different structures in the same collection. However, sometimes, you want to **enforce certain rules** on your data to ensure consistency, such as:

* **Required fields**
* **Field types** (like number, string, date)
* **Data formats**

Even in MongoDB, you can enforce these rules using **Schema Validation** and **Data Validation** features.

# Schema Validation in MongoDB

**Schema Validation** allows you to define **rules** for the structure of your documents.

### Example Use Cases:

* Make sure the email field is always a string and is **required**.
* Ensure the age field is an integer and must be **greater than 0**.
* Ensure that a document always includes certain fields.

### 🔹 How to Set Schema Validation

MongoDB uses **JSON Schema** to define validation rules.

Here’s an example of how to create a collection with **schema validation**:

### Example 1: Defining Validation Rules

db.createCollection("users", {

validator: {

$jsonSchema: {

bsonType: "object",

required: ["name", "email", "age"], // Required fields

properties: {

name: {

bsonType: "string",

description: "Name must be a string"

},

email: {

bsonType: "string",

pattern: "^\\S+@\\S+\\.\\S+$", // Email regex pattern

description: "Email must be a valid email address"

},

age: {

bsonType: "int",

minimum: 18, // Age must be at least 18

description: "Age must be an integer greater than or equal to 18"}}}}});