# Modules

## Module – 1 Introduction to MongoDB

- 1.1 Overview of MongoDB
- 1.2 NoSQL vs. SQL databases
- 1.3 Advantages of using MongoDB
- 1.4 Installing and setting up MongoDB

## Module – 2 MongoDB Data Model

- 2.1 Understanding Documents and Collections
- 2.2 BSON Data Format
- 2.3 Document-Oriented Data Model
- 2.4 MongoDB Schema Design Best Practices

## Module – 3 CRUD Operations in MongoDB

- 3.1 Creating documents
- 3.2 Reading documents
- 3.3 Updating documents
- 3.4 Deleting documents
- 3.5 Querying data with MongoDB

## Module – 4 Indexing and Query Optimization

- 4.1 Importance of indexing
- 4.2 Creating and Managing Indexes
- 4.3 Query Optimization Techniques
- 4.4 Using the Aggregation Framework

## Module – 5 Data Modeling in MongoDB

- 5.1 Embedded vs. Referenced documents
- 5.2 Data normalization and de-normalization
- 5.3 Modeling Realationship

## Module – 6 Advanced MongoDB Features

- 6.1 Geospatial Queries
- 6.2 Text search
- 6.3 Full-Text Search with Text Indexes
- 6.4 Time-Series Data with MongoDB

## Module – 7 MongoDB Atlas and Cloud Deployment

- 7.1 Introduction to MongoDB Atlas
- 7.2 Creating and Managing Clusters
- 7.3 Deploying MongoDB in the Cloud

## Module – 8 Security and Authentication

- 8.1 Securing MongoDB
- 8.2 User Authentication and Authorization
- 8.3 Role-Based Access Control (RBAC)

## Module – 9 Backup and Recovery

- 9.1 Backup Strategies
- 9.2 Restoring Data
- 9.3 Disaster Recovery Planning

## Module – 10 MongoDB Aggregation Pipeline

- 10.1 Aggregation Concepts
- 10.2 Using Pipeline Stages
- 10.3 Custom Aggregation Expressions

## Module – 11 MongoDB Drivers and APIs

- 11.1 MongoDB Drivers
- 11.2 Programming Language of your Choice

## Module – 12 MongoDB Performance Tuning

- 12.1 Profiling and Monitoring
- 12.2 Query Performance Optimization
- 12.3 Hardware and Resource Considerations

## Module – 13 Replication and Sharding

- 13.1 Replication for High Availability
- 13.2 Sharding for Horizontal Scaling
- 13.3 Configuring Replica Sets and Sharded Clusters

## Module – 14 Working with GridFS

- 14.1 Storing Large files in MongoDB
- 14.2 Using GridFS for file Management

# 1

# Introduction to MongoDB

## 1.1 Overview of MongoDB

MongoDB is a popular and widely used open-source, NoSQL database management system. It falls under the category of document-oriented databases and is designed to store, manage, and retrieve data in a flexible and scalable manner. MongoDB is known for its ability to handle large volumes of unstructured or semi-structured data, making it a preferred choice for modern web applications and other data-intensive projects.

MongoDB is often referred to as a "NoSQL" database, which stands for "Not Only SQL." Unlike traditional relational databases like MySQL, PostgreSQL, or Oracle, which use structured tables and SQL (Structured Query Language) for data manipulation, MongoDB uses a flexible and schema-less approach.

In MongoDB, data is stored in BSON (Binary JSON) format, which allows for the storage of diverse data types, such as text, numbers, arrays, and even nested documents, all within a single database collection.

## 1.2 NoSQL vs. SQL databases

To understand MongoDB better, it's essential to compare it with traditional SQL databases:

**1. Data Model**

**SQL:** SQL databases follow a rigid, tabular structure where data is organized into tables with predefined schemas. Data relationships are maintained through foreign keys.

**MongoDB:** MongoDB uses a flexible document-based model, allowing developers to store data in JSON-like documents. Collections (similar to tables) can contain documents with varying structures, making it suitable for dynamic and evolving data.

**2. Scalability**

**SQL:** Scaling SQL databases can be challenging as they typically require vertical scaling (upgrading hardware) or complex sharding solutions for horizontal scaling.

**MongoDB:** MongoDB is designed for horizontal scalability, allowing you to distribute data across multiple servers or clusters easily. This makes it suitable for handling massive amounts of data and high-traffic applications.

**3. Schema**

**SQL:** SQL databases enforce strict schemas, which can be limiting when dealing with evolving data structures.

**MongoDB:** MongoDB does not enforce a fixed schema, offering flexibility in adding or changing fields within documents without affecting existing data.

**4. Query Language**

**SQL:** SQL databases use the SQL query language for data manipulation and retrieval, which is powerful but can be complex for certain tasks.

**MongoDB:** MongoDB uses a query language that is more intuitive for developers familiar with JavaScript, and it supports rich queries, including geospatial and text searches.

## 1.3 Advantages of using MongoDB

MongoDB provides several advantages that make it a popular choice for many applications:

**1. Flexible Schema:** MongoDB's schema-less design allows developers to adapt and evolve their data models as project requirements change over time, without the need for complex migrations.

**2. Scalability:** MongoDB's ability to distribute data across multiple servers or clusters enables seamless horizontal scaling, making it suitable for large-scale and high-traffic applications.

**3. High Performance:** MongoDB's architecture and support for in-memory processing can deliver high-speed read and write operations, which is crucial for responsive applications.

**4. Rich Query Language:** MongoDB offers a powerful and intuitive query language, making it easier to express complex queries, including geospatial and full-text searches.

**5. Automatic Sharding:** MongoDB can automatically distribute data across multiple shards, making it easier to manage and scale databases without manual intervention.

**6. Community and Ecosystem:** MongoDB has a large and active community, which means extensive documentation, a wealth of online resources, and a wide range of third-party libraries and tools.

**7. Document-Oriented:** The document-oriented model of MongoDB closely aligns with the structure of data in many modern applications, simplifying data modeling and reducing impedance mismatch between the application code and the database.

## 1.4 Installing and Setting up MongoDB

Installing MongoDB is the first step to start using it for your projects. Below, I'll outline the general steps for installing and setting up MongoDB:

**1. Choose Your Platform**

MongoDB supports various operating systems, including Windows, macOS, and various Linux distributions. Choose the one that suits your development environment.

**2. Download MongoDB**

Visit the official MongoDB website (https://www.mongodb.com/try/download/community) and download the appropriate installer for your platform. MongoDB offers both a Community Edition (open-source) and a paid Enterprise Edition.

**3. Installation**

Follow the installation instructions for your platform. For most platforms, this involves running the installer and configuring the installation location.

**4. Starting MongoDB**

After installation, you can start MongoDB as a service or as a standalone process, depending on your needs. The exact commands may vary by platform, so refer to MongoDB's documentation for specific instructions.

## 5. Connecting to MongoDB

You can interact with MongoDB using the MongoDB shell, a command-line tool provided with the installation. Use the `mongo` command to connect to your MongoDB instance.

## 6. Create a Database

In MongoDB, databases and collections are created on-the-fly when you insert data. You can create a new database by inserting data into it.

- **Here's a simple** example of installing and setting up MongoDB on a Linux system:

**# Download MongoDB**

wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-4.4.6.tgz

**# Extract the archive**

tar -zxvf mongodb-linux-x86_64-4.4.6.tgz

**# Move MongoDB binaries to a suitable location**

mv mongodb-linux-x86_64-4.4.6 /usr/local/mongodb

**# Start MongoDB as a service**

/usr/local/mongodb/bin/mongod --fork --logpath /var/log/mongodb.log

**# Connect to MongoDB**

/usr/local/mongodb/bin/mongo

**After completing these steps,** you'll have MongoDB up and running on your system, ready for you to create databases, collections, and begin storing and querying data.

# 2

# MongoDB Data Model

## 2.1 Understanding Documents and Collections

In MongoDB, data is organized into two main constructs: documents and collections.

**Documents**

A document in MongoDB is a JSON-like data structure composed of field-value pairs. These documents are the basic unit of data storage and retrieval in MongoDB. Unlike rows in traditional relational databases, MongoDB documents do not require a fixed schema, allowing for flexibility in data structure. Documents can include various data types, including strings, numbers, arrays, and even nested documents. For example, here's a simple MongoDB document representing a user:

- **json**

```
{
  "_id": 1,
  "name": "John Doe",
```

```
  "email": "johndoe@example.com",
  "age": 30,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "zipcode": "12345"
  }
}
```

**Collections**

Collections are containers for MongoDB documents. Each collection can contain zero or more documents, and documents within a collection do not need to have the same structure. Collections can be thought of as analogous to tables in relational databases, but without the rigid schema constraints. For example, you might have a "users" collection to store user documents, a "products" collection for product data, and so on.

## 2.2 BSON Data Format

MongoDB stores data in a binary-encoded format called BSON, which stands for "Binary JSON." BSON extends the capabilities of JSON by adding additional data types and representing complex structures efficiently in binary form. Some of the BSON data types include:

**Double:** For floating-point numbers.

**String:** For text data.

**Boolean:** For boolean values (true or false).

**Object:** For embedded documents.

**Array:** For ordered lists of values.

**Binary Data:** For binary data like images or files.

**ObjectId:** A 12-byte identifier typically used as a unique document identifier.

**Date:** For storing dates and timestamps.

**Null:** Represents a null value.

**Regular Expression:** For storing regular expressions.

The use of BSON allows MongoDB to efficiently store, retrieve, and transmit data, making it a suitable choice for high-performance applications.

## 2.3 Document-oriented Data Model

MongoDB's document-oriented data model provides several advantages, especially for modern application development:

**1. Flexibility:** Documents within a collection can have different structures. This flexibility is especially valuable in scenarios where

the data schema evolves over time. You can add or remove fields without affecting existing documents, making it easy to adapt to changing requirements.

**2. Complex Data Structures:** MongoDB documents support nested arrays and subdocuments, enabling the storage of complex, hierarchical data in a natural way. This is particularly useful for representing deeply nested data, such as product catalogs or organizational hierarchies.

**3. Scalability:** Documents are distributed across multiple servers or clusters, allowing MongoDB to scale horizontally to handle large datasets and high throughput. This scalability is essential for modern, data-intensive applications.

**4. Rich Queries:** MongoDB's query language allows you to perform complex queries, including filtering, sorting, and aggregating data within documents. You can also perform geospatial queries and full-text searches.

**5. No JOINs:** MongoDB does not support JOIN operations like relational databases. Instead, it encourages the de-normalization of data to optimize read performance. While this approach may require more storage space, it can lead to faster queries.

## 2.4 MongoDB Schema Design Best Practices

When designing a schema in MongoDB, there are several best practices to keep in mind:

**1. Data Modeling for Query Performance:** Consider the types of queries your application will perform and design your schema to optimize those queries. Use indexes to speed up query performance for frequently accessed fields.

**2. De-normalization vs. Normalization**: Depending on your use case, you may choose to de-normalize data to avoid complex JOIN operations, or you may normalize data for better consistency. The decision should be based on your application's specific requirements.

**3. Use Embedded Documents Wisely:** Embedding subdocuments within documents can improve query performance by reducing the need for JOINs. However, be cautious about embedding large or frequently updated subdocuments, as they can impact write performance.

**4. Avoid Large Arrays:** Large arrays can become inefficient when elements need to be frequently added or removed. Consider using a separate collection for such scenarios.

**5. Optimize Indexes:** Properly index your collections to speed up queries. Be mindful of the types of queries you'll be performing and create indexes accordingly. Avoid creating too many indexes, as they can impact write performance and consume storage space.

**6. Preallocate Space for Collections:** MongoDB's storage engine allocates space dynamically, but preallocating space for collections can help reduce fragmentation and improve write performance.

**7. Use MongoDB's TTL Index:** If you need to expire data after a certain period, consider using MongoDB's TTL (Time-to-Live) index feature, which automatically removes documents that have reached their expiration date.

**8. Plan for Sharding:** If you anticipate significant data growth, plan for sharding early in your schema design to ensure a smooth scaling process.

**9. Keep Documents Small:** Smaller documents generally result in better write performance and more efficient storage. Avoid including unnecessary data in documents.

**10. Schema Validation:** Use MongoDB's schema validation feature to enforce data consistency and structure within collections.

**Example:**

Let's explain some of these concepts with a simple example. Suppose you are designing a MongoDB schema for an e-commerce application. You might have two collections: "products" and "users."

- **json**

```
// Products Collection
{
  "_id": ObjectId("5fd453fb4e0c103f9cb7f97c"),
  "name": "Smartphone",
  "price": 599.99,
  "category": "Electronics",
  "manufacturer": "Apple",
  "ratings": [4.5, 4.8, 5.0],
  "reviews": [
    {
      "user_id": ObjectId("5fd453fb4e0c103f9cb7f97d"),
      "text": "Great phone!",
      "rating": 5
    },
    {
      "user_id": ObjectId("5fd453fb4e0c103f9cb7f97e"),
```

```
      "text": "Excellent camera",
      "rating": 4
    }
  ]
}
```

**// Users Collection**

```
{
  "_id": ObjectId("5fd453fb4e0c103f9cb7f97d"),
  "name": "Alice",
  "email": "alice@example.com",
  "purchased_products": [
    {
      "product_id": ObjectId("5fd453fb4e0c103f9cb7f97c"),
      "purchase_date": ISODate("2022-05-10T14:30:00Z"),
      "quantity": 1
    }
  ]
}
```

**In this example:**

- The "products" collection uses embedded documents for "reviews" and "ratings," reducing the need for JOINs and improving query performance.
- The "users" collection stores a reference to purchased products using their ObjectId, allowing you to retrieve product details with a separate query when needed.

This simple schema demonstrates how MongoDB's document-oriented data model allows you to store and query data in a way that aligns with your application's requirements.

# 3

# CRUD Operations in MongoDB

CRUD stands for Create, Read, Update, and Delete, and these operations allow you to manage data within MongoDB.

## 3.1 Creating Documents

The **Create** operation in MongoDB involves adding new documents to a collection. A document in MongoDB is a JSON-like structure, and you can insert documents into a collection using the “insertOne()” or “insertMany()” methods.

**“insertOne()”:** This method inserts a single document into a collection. Here's an example:

- **javascript**

```
// Insert a single document into the "products" collection
db.products.insertOne({
  name: "Laptop",
  price: 999.99,
```

```
  category: "Electronics"
});
```

**“insertMany()”:** This method allows you to insert multiple documents into a collection with a single command:

- **javascript**

```
// Insert multiple documents into the "products" collection
db.products.insertMany([
 {
  name: "Keyboard",
  price: 49.99,
  category: "Computer Accessories"
 },
 {
  name: "Mouse",
  price: 19.99,
  category: "Computer Accessories"
 }
]);
```

## 3.2 Reading Documents

The **Read** operation in MongoDB involves retrieving documents from a collection. You can use the “find()” method to query documents in a collection.

**“find()”:** The “find()” method is used to query documents in a collection. It can be used with various options to filter, project, and sort the results. Here's an example:

- **javascript**

```
// Find all documents in the "products" collection
db.products.find();

// Find documents with a specific condition (e.g., price less than $100)
db.products.find({ price: { $lt: 100 } });

// Find documents and project only specific fields (e.g., name and price)
db.products.find({}, { name: 1, price: 1 });
```

## 3.3 Updating Documents

The **Update** operation in MongoDB allows you to modify existing documents in a collection. You can use the “updateOne()” or “updateMany()” methods for this purpose.

**“updateOne()”:** This method updates a single document that matches a specified filter. You can use the `$set` operator to update specific fields:

- **javascript**

```
// Update the price of a specific product
db.products.updateOne(
  { name: "Laptop" },
  { $set: { price: 1099.99 } }
);
```

**“updateMany()”:** This method updates multiple documents that match a specified filter:

- **javascript**

```
// Update prices for all products in the "Electronics" category
db.products.updateMany(
  { category: "Electronics" },
```

```
  { $set: { price: 0.9 * price } }
);
```

## 3.4 Deleting Documents

The **Delete** operation in MongoDB allows you to remove documents from a collection. You can use the “deleteOne()” or “deleteMany()” methods.

**“deleteOne()”:** This method deletes a single document that matches a specified filter:

- **javascript**

```
// Delete a specific product document
db.products.deleteOne({ name: "Mouse" });
```

**“deleteMany()”:** This method deletes multiple documents that match a specified filter:

- **javascript**

```
// Delete all products with a price less than $50
db.products.deleteMany({ price: { $lt: 50 } });
```

## 3.5 Querying Data with MongoDB

MongoDB provides a powerful query language that allows you to filter and retrieve data based on specific criteria. Here are some commonly used query operators:

**Comparison Operators:** You can use operators like “$eq”, “$ne”, “$gt”, “$lt”, “$gte”, and “$lte” to compare values:

- **javascript**

```
// Find products with a price greater than $500
db.products.find({ price: { $gt: 500 } });
```

**Logical Operators:** MongoDB supports logical operators like “$and”, “$or”, and “$not” for combining conditions:

- **javascript**

```
// Find products that are either in the "Electronics" category or have a price less than $50
db.products.find({
  $or: [
    { category: "Electronics" },
    { price: { $lt: 50 } }
```

```
  ]
});
```

**Array Operators:** You can query arrays using operators like “$in”, “$nin”, “$all”, and “$elemMatch”:

- **javascript**

```
// Find products with specific tags in the "tags" array
db.products.find({ tags: { $in: ["gaming", "wireless"] } });

// Find products with all specified tags in the "tags" array
db.products.find({ tags: { $all: ["electronics", "accessories"] } });
```

**Regular Expressions:** MongoDB allows you to perform text searches using regular expressions:

- **javascript**

```
// Find products with names containing "laptop" (case-insensitive)
db.products.find({ name: /laptop/i });
```

**Projection:** You can specify which fields to include or exclude in query results using projection:

- **javascript**

```
// Find products and include only the "name" and "price" fields in the results
db.products.find({}, { name: 1, price: 1 });
```

**Sorting:** MongoDB allows you to sort query results based on one or more fields:

- **javascript**

```
// Find products and sort them by price in ascending order
db.products.find().sort({ price: 1 });
```

These are just some examples of the powerful querying capabilities provided by MongoDB. MongoDB's rich query language enables you to retrieve precisely the data you need from your collections.

**Example:**

Let's put these CRUD operations and querying capabilities into practice with an example. Suppose you have a "users" collection that stores information about users of an online platform:

- **json**

```
// Sample "users" collection
[
 {
  "_id": ObjectId("5fd453fb4e0c103f9cb7f97d"),
  "name": "Alice",
  "email": "alice@example.com",
  "age": 28
 },
 {
  "_id": ObjectId("5fd453fb4e0c103f9cb7f97e"),
  "name": "Bob",
  "email": "bob@example.com",
  "age": 32
 },
 {
  "_id": ObjectId("5fd453fb4e0c103f9cb7f97f"),
  "name": "Charlie",
  "email": "charlie@example.com",
  "age": 24
 }
```

]

Now, let's perform some CRUD operations and queries:

**Create:**

**Add a new user:**

- **javascript**

```
db.users.insertOne({
  name: "David",
  email: "david@example.com",
  age:

30
});
```

**Read:**

**Retrieve all users:**

- **javascript**

```
db.users.find();
```

**Retrieve users younger than 30:**

- **javascript**

db.users.find({ age: { $lt: 30 } });

**Update:**

**Update Bob's age:**

javascript

db.users.updateOne(

{ name: "Bob" },

{ $set: { age: 33 } }

);

```

**Delete:**

**Delete Charlie's record:**

- **javascript**

db.users.deleteOne({ name: "Charlie" });
```

**Query:**

**Find users with email addresses containing "example.com":**

- **javascript**

```
db.users.find({ email: /example\.com/ });
```

**Find users older than 25 and sort them by age in descending order:**

- **javascript**

```
db.users.find({ age: { $gt: 25 } }).sort({ age: -1 });
```

**These examples** demonstrate how to perform CRUD operations and queries in MongoDB, showcasing the versatility and flexibility of the database for managing and retrieving data.