# MongoDB Notes

## No Schema

"No Schema" refers to the flexibility it provides in terms of document structure within a collection. Unlike traditional relational databases where you define a fixed schema with tables, columns, and relationships, MongoDB allows you to store documents in collections without enforcing a rigid schema.

This means that documents within a collection can have varying structures, and fields can be added, modified, or removed dynamically without the need to define a schema beforehand. Each document can have its own unique structure, with different fields and data types.

## How MongoDB Works

MongoDB is a document-oriented database, classified as a NoSQL database. It stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time. MongoDB doesn't require a schema to be defined before adding data; however, you can enforce a schema if needed.

MongoDB works on the principle of collections and documents. A collection is a grouping of MongoDB documents and is the equivalent of a table in a relational database. Documents are individual records within a collection and are similar to rows in a table.

## Insert First Data

To insert data into MongoDB, you can use the insertOne() method. Here's a basic example:

```
db.collection("myCollection").insertOne({ "name": "John", "age": 30 });
```

This command inserts a document with the fields "name" and "age" into the "myCollection" collection.

## CRUD Operations

CRUD stands for Create, Read, Update, Delete. MongoDB provides methods to perform these operations:

- Create: insertOne() or insertMany() to add new documents.
- Read: find() to retrieve documents.
- Update: updateOne() or updateMany() to modify existing documents.
- Delete: deleteOne() or deleteMany() to remove documents.

## Insert Many

To insert multiple documents at once, you can use the insertMany() method. Here's an example:

```
db.collection("myCollection").insertMany([
   { "name": "Alice", "age": 25 },
   { "name": "Bob", "age": 35 },
   { "name": "Charlie", "age": 40 }
]);
```

This inserts three documents into the "myCollection" collection.

**Update and Update Many**

**Update:** The updateOne() method in MongoDB updates the first document that matches the specified criteria.

To update documents, you can use the updateOne() or updateMany() method. Here's an example:

Ex1:

```
db.collection("myCollection").updateOne(
   { "name": "Alice" }, // Filter
  { $set: { "age": 26 }} // Update
);
```

This updates the age of the document where the name is "Alice" to 26.

**Update Many**: The updateMany() method updates all documents that match the specified criteria

Ex2:

```
db.employees.updateMany(
  { "jobTitle": "Software Engineer" }, // Filter criteria
  { $set: { "department": "Engineering" }} // Update operation
);
```

All documents where the job title is "Software Engineer" will have their department updated to "Engineering".

**Delete and Delete Many**

To delete documents, you can use the deleteOne() or deleteMany() method. Here's an example:

Ex1:

```
db.collection("myCollection").deleteOne({ "name": "Alice" });
```

This deletes the document where the name is "Alice" from the collection.

Ex2:

```
db.employees.deleteMany(
  { "age": { $gt: 50 } } // Filter criteria
);
```

this operation, all documents where the age is greater than 50 will be deleted from the collection.

**Projection:**

Projection in MongoDB allows you to specify which fields to return in the query result. It's helpful for optimizing performance and bandwidth usage

```
db.collection.find(
  <query>,      // Criteria to match documents
  <projection>  // Fields to include/exclude
)
```

Example:
// Include only specified fields
```
db.collection.find({}, { name: 1, age: 1 })
```

// Exclude specified fields
```
db.collection.find({}, { _id: 0, name: 1 })
```

**Introduction to Embedded Documents:**
Embedded documents in MongoDB allow you to nest documents within one another. This is useful for representing complex data structures.

Example:
```
{
  name: "John Doe",
  address: {
    street: "123 Main St",
    city: "Anytown",
    country: "USA"
  }
}
```

**Embed Documents in Action:**
You can embed documents directly into other documents when inserting or updating data.
```
db.users.insertOne({
  name: "Alice",
  address: {
    street: "456 Elm St",
    city: "Sometown",
    country: "Canada"
  }
})
```

**Adding Arrays:**
MongoDB supports arrays as field values, allowing you to store multiple values within a single field.
Example:
```
db.students.insertOne({
  name: "Bob",
  subjects: ["Math", "Science", "History"]
})
```

**Fetching Data From Structured Data:**
MongoDB's flexible schema allows you to fetch data based on structured or unstructured criteria.
Example:
```
db.students.find({ "address.city": "Sometown" })
```

**Aggregation:**
Aggregation in MongoDB allows you to perform operations like grouping, filtering, and transforming data.
Example:

```
db.sales.aggregate([
  { $group: { _id: "$product", total: { $sum: "$amount" } } }
])
```

**Schema Types:**
MongoDB supports various data types for fields within documents. Some common types include:

1. **String:** Used for textual data.
2. **Number:** Used for numeric data.
3. **Boolean:** Used for boolean values (true/false).
4. **Date:** Used for storing dates.
5. **Array:** Used for storing arrays of values.
6. **Object:** Used for nested objects/documents.
7. **ObjectId:** Used to store unique identifiers.
8. **Binary Data:** Used for storing binary data.
9. **Null:** Used to represent null values.
10. **Regular Expression:** Used to store regular expressions.
11. **Geospatial:** Used for storing geospatial data.
12. **Decimal128:** Used for storing high precision floating-point numbers.

**Types of Data in MongoDB:**
MongoDB can store structured, semi-structured, and unstructured data. This includes:

1. **Structured Data:** Data organized in a well-defined schema.
2. **Semi-Structured Data**: Data with some organization but without a strict schema.
3. **Unstructured Data:** Data without a predefined schema, such as documents, images, videos, etc.

**Relationship between Data:**
In MongoDB, relationships between data can be established in different ways:

1. **Embedding:** Nesting one document within another.
2. **Referencing:** Storing references (usually _id fields) to related documents.

**One-to-One Relationship using Embedding:**
In a one-to-one relationship using embedding, one document is nested within another document.

Example:
// User document with embedded address

```
{
  _id: ObjectId("user_id"),
  name: "John",
  address: {
    street: "123 Main St",
    city: "Anytown",
    country: "USA"
  }
}
```

**One-to-Many Relationship using Embedding:**
In a one-to-many relationship using embedding, multiple documents are nested within another document.

Example:
// Author document with embedded books
```
{
  _id: ObjectId("author_id"),
  name: "J.K. Rowling",
  books: [
    { title: "Harry Potter and the Sorcerer's Stone", year: 1997 },
    { title: "Harry Potter and the Chamber of Secrets", year: 1998 }
  ]
}
```

**One-to-One Relationship using Referencing:**
In a one-to-one relationship using referencing, one document contains a reference to another document.

Example:
// User document with reference to address
```
{
  _id: ObjectId("user_id"),
  name: "Alice",
  address: ObjectId("address_id")
}
```

// Address document
```
{
  _id: ObjectId("address_id"),
  street: "456 Elm St",
  city: "Sometown",
  country: "Canada"
}
```

One-to-Many Relationship using Referencing:
In a one-to-many relationship using referencing, one document contains references to multiple related documents.
// Author document with references to books
```
{
  _id: ObjectId("author_id"),
  name: "George R.R. Martin",
  books: [ObjectId("book_id1"), ObjectId("book_id2")]
}
```

// Book documents
```
{
  _id: ObjectId("book_id1"),
  title: "A Game of Thrones",
  year: 1996
}
```

```
{
  _id: ObjectId("book_id2"),
  title: "A Clash of Kings",
  year: 1998
}
```