Basics of MongoDB

Introduction to MongoDB .....

📌 What is MongoDB?

MongoDB is a **NoSQL database** designed for **high performance, scalability, and flexibility**. Unlike traditional relational databases (SQL-based), it stores data in a **document-oriented** format (JSON-like BSON).

✅ **Key Features:**

- **Schema-less:** No fixed structure; you can store different fields in different documents.
- **Scalability:** Supports horizontal scaling using **sharding**.
- **High Availability:** Uses **replication** for fault tolerance.
- **Rich Query Language:** Supports powerful queries with filtering, aggregation, and indexing.

📌 NoSQL vs. SQL Databases

SQL Databases:

- Structured data stored in tables with rows and columns.
- Fixed schema (predefined structure).
- Examples: MySQL, PostgreSQL, Oracle.

NoSQL Databases:

- Unstructured or semi-structured data stored in collections of documents.
- Dynamic schema (flexible structure).
- Examples: MongoDB, Cassandra, Redis.

Key Differences:

- **Scalability**: NoSQL databases are highly scalable for large datasets.
- **Flexibility**: NoSQL databases allow for flexible schema design.
- **Transactions**: SQL databases support ACID transactions, while NoSQL databases like MongoDB support multi-document transactions (since v4.0).

✅ **MongoDB is ideal for:**
✔ **Big Data & Real-time Applications** (e.g., IoT, AI)
✔ **Scalable Applications** (e.g., E-commerce, Social Media)
✔ **Flexible Schema Requirements** (e.g., CMS, Logs, Analytics)

📌 Advantages & Use Cases of MongoDB

✅ Advantages:

- **Flexible Schema:** No need for predefined schemas.
- **High Performance:** Uses **indexes** and in-memory caching for speed.
- **Scalability:** Supports **sharding** for handling large datasets.
- **Replication:** Ensures **high availability** via replica sets.
- **Developer-Friendly:** JSON-like data format simplifies usage with JavaScript, Node.js, Python, etc.

✅ Use Cases:

✔ **E-commerce** (Products, Orders, Transactions)
✔ **Social Media** (Users, Posts, Comments, Likes)
✔ **Real-time Analytics** (IoT, Streaming, Logs)
✔ **Content Management Systems (CMS)**
✔ **Location-based Services** (Geospatial Queries)

📌 Installation & Setup of MongoDB

1️⃣ Install MongoDB (Community Edition)

- **Windows:** Download from MongoDB official site
- **Mac (Homebrew):**

sh Copy Edit

```
brew tap mongodb/brew
brew install mongodb-community@7.0
```

- **Linux (Ubuntu/Debian):**

sh Copy Edit

```
sudo apt update
sudo apt install -y mongodb
```

2️⃣ Start the MongoDB Server

- **Windows:** Run `mongod.exe` from `C:\Program Files\MongoDB\Server\bin`
- **Mac/Linux:**

sh Copy Edit

```
sudo systemctl start mongod
```

3️⃣ Verify MongoDB is Running

Run:

sh Copy Edit

```
mongo --version
```

4️⃣ Start the MongoDB Shell (Legacy) or `mongosh` (Modern)

sh Copy Edit

```
mongosh
```

This opens an interactive MongoDB shell where you can run commands.

MongoDB Basics .....

📌 JSON vs. BSON

🔹 What is JSON?

- JSON (**JavaScript Object Notation**) is a lightweight data format used for data exchange. It is human-readable and widely used in web applications.

Example JSON Document:

```
{
  "name": "Alice",
  "age": 25,
  "email": "alice@example.com",
  "hobbies": ["reading", "traveling"]
}
```

🔹 What is BSON?

- BSON (**Binary JSON**) is the internal data format used by MongoDB. It is similar to JSON but supports **additional data types** like `ObjectId`, `Date`, and `Binary Data`.

Example BSON Representation:

```
{
  "_id": ObjectId("65bf9a1c9b3f6b4aef987654"),
  "name": "Alice",
  "age": 25,
  "email": "alice@example.com",
  "hobbies": ["reading", "traveling"],
  "createdAt": ISODate("2024-02-01T10:30:00Z")
}
```

✅ Key Differences Between JSON & BSON:

Feature	JSON	BSON
Storage	Text-based (human-readable)	Binary (optimized for MongoDB)
Speed	Slower (parsing large JSON is slow)	Faster (binary format is optimized)
Data Types	Limited (String, Number, Boolean, Array)	Supports extra types (ObjectId, Date, Binary)
Size	Compact	Slightly larger due to metadata

📌 Data Types in MongoDB

Data Types in MongoDB

- **String**: UTF-8 encoded text.
    - Example: "name": "John"
- **Number**: Can be an integer (32-bit or 64-bit) or a floating-point number.
    - Example: "age": 30
- **Boolean**: true or false.
    - Example: "isStudent": false
- **Date**: Stores date and time in ISO format.
    - Example: "birthdate": new Date("1990-01-01")
- **Array**: A list of values.
    - Example: "hobbies": ["reading", "swimming", "coding"]
- **Object**: Embedded documents (nested JSON/BSON).
    - Example:

```
"address": {
  "city": "New York",
  "zip": "10001"
}
```

- **ObjectId**: A unique identifier for documents (12-byte hexadecimal).
    - Example: "_id": ObjectId("507f1f77bcf86cd799439011")
- **Null**: Represents a null value.
    - Example: "middleName": null
- **Binary Data**: Stores binary data (e.g., images, files).
    - Example: "image": BinData(0, "base64encodeddata")
- **Timestamp**: Internal use for tracking document modifications.
    - Example: "timestamp": Timestamp(1633072800, 1)
- **Regular Expression**: Stores regex patterns.
    - Example: "pattern": /^[A-Za-z]+$/

Example Document with Different Data Types:

```
{
  "_id": ObjectId("65bf9a1c9b3f6b4aef987654"),
  "name": "Alice",
  "age": 25,
  "isActive": true,
  "email": "alice@example.com",
  "createdAt": ISODate("2024-02-01T10:30:00Z"),
  "hobbies": ["reading", "traveling"],
  "address": {
    "city": "New York",
    "zip": "10001"
  },
  "deleted": null
}
```

📌 Understanding Collections and Documents

🔹 What is a Document?

A **document** in MongoDB is a **JSON-like object** that stores **data**.

- Similar to a **row** in a relational database.
- Each document is **independent** and can have a **different structure** from others in the same collection.

Example Document:

```
{
  "_id": ObjectId("507f1f77bcf86cd799439011"),
  "name": "John",
  "age": 30,
  "address": {
    "city": "New York",
    "zip": "10001"
  }
}
```

🔹 What is a Collection?

A **collection** is a **group of documents**, similar to a **table** in relational databases.

- **Documents inside a collection do not need to have the same structure**.
- A **collection is created automatically** when you insert a document into it.

Example Collection (`users`):

```
users
├── { "_id": ObjectId("65bf9a1c9b3f6b4aef987654"), "name": "Alice", "email": "alice@example.com" }
├── { "_id": ObjectId("65bf9a1c9b3f6b4aef987655"), "name": "Bob", "email": "bob@example.com" }
├── { "_id": ObjectId("65bf9a1c9b3f6b4aef987656"), "name": "Charlie", "email": "charlie@example.com" }
```

📌 Schema-less Nature of MongoDB

MongoDB is **schema-less**, meaning:
✔ You **don't need to define a fixed schema** before inserting data.
✔ Each document in a collection **can have different fields**.
✔ **Flexible** for storing unstructured or semi-structured data.

Example: Different Documents in the Same Collection

```
// Document 1
{
  "name": "John",
  "age": 30
}

// Document 2
{
  "name": "Jane",
  "isStudent": true,
  "hobbies": ["reading", "swimming"]
}
```

Advantages of Schema-less Design:

- **Flexibility**: Easily adapt to changing data requirements.
- **Rapid Development**: No need to define and update schemas upfront.
- **Scalability**: Suitable for unstructured or semi-structured data.

Disadvantages:

- **Data Integrity**: No enforcement of data structure or types.
- **Complexity**: Harder to maintain consistency across documents.

Schema Validation:

- MongoDB allows optional schema validation rules to enforce structure and data types.

- Example:

```
db.createCollection("users", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "age"],
      properties: {
        name: { bsonType: "string" },
        age: { bsonType: "int" }
      }
    }
  }
})
```

CRUD Operations .....

Operation	Method	Example
Create	`insertOne()`	`db.users.insertOne({ name: "Alice" })`
Create	`insertMany()`	`db.users.insertMany([{ name: "Bob" }, { name: "Charlie" }])`
Read	`findOne()`	`db.users.findOne({ name: "Alice" })`
Read	`find()`	`db.users.find({ age: { $gt: 25 } })`
Update	`updateOne()`	`db.users.updateOne({ name: "Alice" }, { $set: { age: 26 } })`
Update	`updateMany()`	`db.users.updateMany({ age: { $lt: 30 } }, { $set: { status: "active" } })`
Delete	`deleteOne()`	`db.users.deleteOne({ name: "Charlie" })`
Delete	`deleteMany()`	`db.users.deleteMany({ age: { $gt: 50 } })`

📌 1. Create Operations

These operations are used to insert documents into a collection.

- `insertOne()`:
  - Inserts a single document into a collection.
  - Example:

```javascript
db.users.insertOne({
  name: "John",
  age: 30,
  isStudent: false
});
```

  - Output:

```json
{
  "acknowledged": true,
  "insertedId": ObjectId("507f1f77bcf86cd799439011")
}
```

- `insertMany()`:
  - Inserts multiple documents into a collection.
  - Example:

```javascript
db.users.insertMany([
  { name: "Alice", age: 25, isStudent: true },
  { name: "Bob", age: 35, isStudent: false }
]);
```

  - Output:

```json
{
  "acknowledged": true,
  "insertedIds": [
    ObjectId("507f1f77bcf86cd799439012"),
    ObjectId("507f1f77bcf86cd799439013")
  ]
}
```

📌 2. Read Operations

These operations are used to query documents from a collection.

- `find()`:
  - Retrieves all documents that match the query criteria.
  - Example:

```javascript
db.users.find({ isStudent: true });
```

  - Output:

```json
[
  { "_id": ObjectId("507f1f77bcf86cd799439012"), "name": "Alice", "age": 25, "isStudent": true }
]
```

- `findOne()`:
  - Retrieves the first document that matches the query criteria.
  - Example:

```javascript
db.users.findOne({ age: { $gt: 30 } });
```

  - Output:

```json
{
  "_id": ObjectId("507f1f77bcf86cd799439013"),
  "name": "Bob",
  "age": 35,
  "isStudent": false
}
```

Filters:

Use query operators to filter documents.

- Common operators:
    - ★ `$eq`: Equal to.
    - ★ `$ne`: Not equal to.
    - ★ `$gt`: Greater than.
    - ★ `$lt`: Less than.
    - ★ `$in`: Matches any value in an array.
    - ★ `$regex`: Matches a regular expression.

- Example:

```javascript
db.users.find({ age: { $gt: 25, $lt: 40 } });
```

📌 3. Update Operations

These operations are used to modify existing documents in a collection.

- `updateOne()`:
  - Updates the first document that matches the query criteria.
  - Example:

```javascript
db.users.updateOne(
  { name: "John" },
  { $set: { age: 31 } }
);
```

  - Output:

```json
{
  "acknowledged": true,
  "matchedCount": 1,
  "modifiedCount": 1
}
```

- `updateMany()`:
  - Updates all documents that match the query criteria.
  - Example:

```javascript
db.users.updateMany(
  { isStudent: true },
  { $set: { isStudent: false } }
);
```

  - Output:

```json
{
  "acknowledged": true,
  "matchedCount": 2,
  "modifiedCount": 2
}
```

Update Operators:

Operator	Description	Example
`$set`	Updates/sets a field	`{ $set: { age: 30 } }`
`$unset`	Removes a field	`{ $unset: { email: "" } }`
`$inc`	Increments a value	`{ $inc: { age: 1 } }`
`$push`	Adds value to an array	`{ $push: { hobbies: "coding" } }`
`$pull`	Removes value from an array	`{ $pull: { hobbies: "reading" } }`

★ **`$set`**: Sets the value of a field.

★ **`$unset`**: Removes a field from a document.

★ **`$inc`**: Increments the value of a field.

★ **`$push`**: Adds an element to an array.

★ **`$pull`**: Removes an element from an array.

- Example:

javascript Copy

```javascript
db.users.updateOne(
  { name: "John" },
  { $inc: { age: 1 }, $push: { hobbies: "reading" } }
);
```

📌 4. Delete Operations

These operations are used to remove documents from a collection.

- `deleteOne()`:
  - Deletes the first document that matches the query criteria.
  - Example:

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

  - Output:

```json
{
  "acknowledged": true,
  "deletedCount": 1
}
```

- `deleteMany()`:
  - Deletes all documents that match the query criteria.
  - Example:

```javascript
db.users.deleteMany({ isStudent: false });
```

  - Output:

```json
{
  "acknowledged": true,
  "deletedCount": 2
}
```

Querying in MongoDB .....

Querying is essential to efficiently retrieve data from a MongoDB collection. Let's explore how to filter, sort, project, paginate, and use regex in queries.

Feature	Method	Example
Filtering	`$gt`, `$lt`, `$eq`, `$in`	`{ age: { $gt: 25 } }`
Sorting	`.sort({ field: 1 or -1 })`	`.sort({ age: -1 })`
Projection	Select fields	`.find({}, { name: 1, email: 1, _id: 0 })`
Pagination	`.limit(n)`, `.skip(n)`	`.skip(5).limit(5)`
Regex	`/pattern/`	`{ name: /John/i }`

📌 1. Filtering Using Operators

MongoDB provides several query operators to filter data. Here are some commonly used ones:

Comparison Operators

Operator	Description	Example
`$eq`	Matches a specific value	`{ age: { $eq: 25 } }`
`$ne`	Not equal to	`{ status: { $ne: "inactive" } }`
`$gt`	Greater than	`{ age: { $gt: 30 } }`
`$lt`	Less than	`{ age: { $lt: 30 } }`
`$gte`	Greater than or equal to	`{ age: { $gte: 18 } }`
`$lte`	Less than or equal to	`{ age: { $lte: 50 } }`
`$in`	Matches any value in an array	`{ status: { $in: ["active", "pending"] } }`
`$nin`	Excludes values in an array	`{ age: { $nin: [18, 21, 25] } }`

📌 Example: Filtering users older than 25

javascript Copy

```javascript
db.users.find({ age: { $gt: 25 } }).pretty();
```

📌 Example: Get users whose status is either "active" or "pending"

javascript Copy

```javascript
db.users.find({ status: { $in: ["active", "pending"] } }).pretty();
```

Logical Operators

Operator	Description	Example
`$eq`	Matches a specific value	`{ age: { $eq: 25 } }`
`$ne`	Not equal to	`{ status: { $ne: "inactive" } }`
`$gt`	Greater than	`{ age: { $gt: 30 } }`
`$lt`	Less than	`{ age: { $lt: 30 } }`
`$gte`	Greater than or equal to	`{ age: { $gte: 18 } }`
`$lte`	Less than or equal to	`{ age: { $lte: 50 } }`
`$in`	Matches any value in an array	`{ status: { $in: ["active", "pending"] } }`
`$nin`	Excludes values in an array	`{ age: { $nin: [18, 21, 25] } }`

📌 **Example: Find users who are either inactive or older than 50**

javascript Copy

```javascript
db.users.find({
  $or: [{ age: { $gt: 50 } }, { status: "inactive" }]
}).pretty();
```

📌 **Example: Find users who are active and older than 25**

javascript Copy

```javascript
db.users.find({
  $and: [{ age: { $gt: 25 } }, { status: "active" }]
}).pretty();
```

📌 2. Sorting Results

You can sort query results using the .sort() method.

📌 **Syntax:**

javascript Copy

```javascript
db.collection.find().sort({ field: 1 or -1 });
```

- `1` → Ascending order
- `-1` → Descending order

📌 **Example: Get users sorted by age in ascending order**

```javascript
db.users.find().sort({ age: 1 }).pretty();
```

📌 **Example: Get users sorted by name in descending order**

```javascript
db.users.find().sort({ name: -1 }).pretty();
```

📌 3. Projection (Selecting Specific Fields)

By default, MongoDB returns all fields in a document. You can select specific fields using projection.

📌 **Syntax:**

```javascript
db.collection.find({ query }, { field1: 1, field2: 1, _id: 0 });
```

- `1` → Include the field
- `0` → Exclude the field

📌 **Example: Get only names and emails of users (excluding `_id`)**

```javascript
db.users.find({}, { name: 1, email: 1, _id: 0 }).pretty();
```

📌 **Example: Get all users but exclude the `email` field**

```javascript
db.users.find({}, { email: 0 }).pretty();
```

📌 4. Pagination (limit() & skip())

Pagination is crucial for handling large datasets efficiently.

📌 **Syntax:**

```javascript
db.collection.find().skip(offset).limit(number);
```

- `.skip(n)` → Skips the first `n` documents
- `.limit(n)` → Limits the results to `n` documents

📌 **Example: Get the first 5 users**

```javascript
db.users.find().limit(5).pretty();
```

📌 **Example: Get the next 5 users (pagination: page 2, assuming 5 per page)**

```javascript
db.users.find().skip(5).limit(5).pretty();
```

📌 **Tip: To implement pagination in Node.js, use:**

```javascript
const page = 2;
const limit = 5;
const skip = (page - 1) * limit;

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

📌 5. Regular Expressions in Queries

MongoDB supports regex (Regular Expressions) for searching patterns in strings.

📌 **Example: Find users whose name starts with "A"**

```javascript
db.users.find({ name: /^A/ }).pretty();
```

📌 **Example: Find users whose email contains "gmail"**

```javascript
db.users.find({ email: /gmail/ }).pretty();
```

📌 **Example: Find users whose name ends with "son"**

```javascript
db.users.find({ name: /son$/ }).pretty();
```

📌 **Example: Case-Insensitive Search (using `/pattern/i`)**

```javascript
db.users.find({ name: /alice/i }).pretty();
```

Indexing in MongoDB .....

Indexing is essential for improving query performance in MongoDB.

Types of Indexes: Single Field, Compound, Multikey, Text, Geospatial, Unique, TTL, Sparse.

Use `getIndexes()` to view existing indexes on a collection.

Best Practices: Create indexes for frequently queried fields, avoid over-indexing, and monitor index usage.

Index Type	Use Case
Single Field Index	Optimizing queries on a single field
Compound Index	Optimizing queries with multiple fields
Multikey Index	Optimizing queries on array fields
Text Index	Full-text search (e.g., product descriptions, blog posts)
Geospatial Index	Location-based queries (e.g., nearby places, GPS data)

📌 1. What is Indexing?

An **index** is a **data structure** that stores a small portion of a collection's data in an **organized** manner to **speed up queries**.

- 🔹 **Without an index**, MongoDB **performs a full collection scan**, which is slow.
- 🔹 **With an index**, MongoDB can **quickly locate** documents based on indexed fields.

📌 **Example:** If we have 1 million users and search for `{ age: 30 }`:

- **Without Index:** MongoDB checks **all** 1 million documents.
- **With Index:** MongoDB finds **only relevant** documents efficiently.

```
db.users.createIndex({ age: 1 }); // Index on 'age' in ascending order
```

📌 2. Types of Indexes in MongoDB

1) Single Field Index (Basic Indexing)

- 🔹 **Created on a single field** to speed up searches.

Syntax:

```
db.collection.createIndex({ fieldName: 1 }); // Ascending
db.collection.createIndex({ fieldName: -1 }); // Descending
```

Example: Create an index on the `email` field

```
db.users.createIndex({ email: 1 });
```

Queries using `email` will be much faster!

2) Compound Index (Index on Multiple Fields)

- 🔹 **Indexes multiple fields together** to optimize queries using multiple conditions.

Syntax:

```
db.collection.createIndex({ field1: 1, field2: -1 });
```

Example: Index on `age` (asc) and `status` (desc)

```
db.users.createIndex({ age: 1, status: -1 });
```

Optimized for queries like:

```
db.users.find({ age: 25, status: "active" });
```

Compound Index Tip:

- Queries should **follow the order** of indexed fields.
- Index `{ age: 1, status: -1 }` is **optimized for queries using `age` first**.

3) Multikey Index (For Arrays)

- Indexes array fields, allowing searches inside arrays.
- Automatically created when indexing an array field.

Example: Index on an array field `tags`

```
db.products.createIndex({ tags: 1 });
```

Optimized for queries like:

```
db.products.find({ tags: "electronics" });
```

Use Case: Finding documents with an array field (e.g., blog categories, product tags).

4) Text Index (For Full-Text Search)

- Used for searching text fields efficiently.
- Supports case-insensitive and partial word matches.

Example: Create a text index on `description` field

```
db.products.createIndex({ description: "text" });
```

Optimized for text search queries:

```
db.products.find({ $text: { $search: "laptop" } });
```

Use Case: Searching product descriptions, blog posts, reviews, etc.

5) Geospatial Index (For Location-Based Queries)

- Used for storing and querying location data (`latitude`, `longitude`).
- Supports `2dsphere` for Earth-like geometry and `2d` for flat geometry.

Example: Create a geospatial index on `location` field

```
db.places.createIndex({ location: "2dsphere" });
```

Optimized for location-based queries:

```
db.places.find({
  location: {
    $near: {
      $geometry: { type: "Point", coordinates: [40.7128, -74.0060] }, // NYC coordinates
      $maxDistance: 5000 // 5km range
    }
  }
});
```

Use Case: Finding nearby restaurants, hotels, or users within a location.

Other Index Types

📌 **Other Index Types**

- **Unique Index**: Ensures that the indexed field(s) have unique values.

javascript Copy

```
db.users.createIndex({ email: 1 }, { unique: true });
```

- **TTL Index**: Automatically removes documents after a specified time.

javascript Copy

```
db.logs.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 }); // Expires after 1 hour
```

- **Sparse Index**: Only indexes documents that contain the indexed field.

javascript Copy

```
db.users.createIndex({ middleName: 1 }, { sparse: true });
```

📌 3. How Indexing Improves Query Performance

- 🔹 Indexes **significantly improve query performance** by reducing the number of documents scanned.

Without Index:

- MongoDB performs a **collection scan**, scanning every document in the collection.
- This is slow and inefficient for large datasets.

With Index:

- MongoDB uses the index to quickly locate the relevant documents.
- Reduces the number of documents scanned, improving query performance.

Example:

- Query: db.users.find({ age: { $gt: 25 } })
- Without Index: Scans all documents in the users collection.
- With Index: Uses the age index to quickly find documents where age > 25.

Example: Querying without an index (slow)

```
db.users.find({ email: "user@example.com" }).explain("executionStats");
```

Result: `"COLLSCAN"` (Collection Scan) → **Slow query** 🚨

Example: Querying with an index (fast)

```
db.users.createIndex({ email: 1 });
db.users.find({ email: "user@example.com" }).explain("executionStats");
```

Result: `"IXSCAN"` (Index Scan) → **Optimized query** ⚡

📌 4. Checking Existing Indexes

- 🔹 Use the getIndexes() method to view all indexes on a collection.

Syntex:

```
db.collection.getIndexes();
```

Example:

```
db.users.getIndexes();
```

Output:

```
[
  {
    "v": 2,
    "key": { "_id": 1 }, // Default index on _id
    "name": "_id_"
  },
  {
    "v": 2,
    "key": { "age": 1 }, // User-created index on age
    "name": "age_1"
  }
]
```

📌 5. Indexing Best Practices

- **Create Indexes for Frequently Queried Fields**:
  - Identify slow queries using `.explain()` and create indexes for the fields involved.
- **Avoid Over-Indexing**:
  - Too many indexes can slow down write operations (inserts, updates, deletes).
- **Use Compound Indexes Wisely**:
  - Order of fields in a compound index matters. Place the most selective fields first.
- **Monitor Index Usage**:
  - Use the `$indexStats` aggregation stage to track index usage.

javascript Copy

```
db.users.aggregate([{ $indexStats: {} }]);
```

📌 6. Example: Creating and Using an Index

1. **Create an Index**:

```javascript
db.users.createIndex({ age: 1 }); // Create an ascending index on age
```

2. **Query Using the Index**:

```javascript
db.users.find({ age: { $gt: 25 } }).explain("executionStats");
```

3. **Check Index Usage**:

```javascript
db.users.getIndexes();
```