MongoDB

What is MongoDB?

MongoDB is a NoSQL database. SQL databases handle structured data in rows and columns. NoSQL databases, on the other hand, are non-relational databases designed to handle large volumes of data, and they do so differently from traditional databases like SQL.

Key Concepts

Document-Oriented: In MongoDB, data is stored in documents, which are like rows in a table in SQL. Each document can have a different structure, allowing you to store flexible and unstructured data.

Collections: Documents are grouped into collections, which are similar to tables in SQL. Collections store related data, but the documents within a collection can have varying structures.

JSON (JavaScript Object Notation): A lightweight data exchange format that is easy for developers to read and write. Instead of adhering to a specific order like tabular rows and columns, JSON allows for the exchange of data structures in a way that is easy for developers to read and write.

Benefits

Scalability: MongoDB can handle large volumes of data and traffic, making it suitable for applications that need to scale.

Flexibility: Its document-based structure allows you to adapt to changing data requirements easily.

Performance: MongoDB can be really fast for certain types of queries.

Community: There's a large and active MongoDB community, so you can find plenty of resources and support.

Installation

1. Go to this link https://www.mongodb.com/try/download/community.
2. Select your platform (Mac, Windows, Linux) and the appropriate version for your platform.
3. Click the “Download” button. Keep in mind that downloads vary for each browser and operating system.
4. Install MongoDB. Example for each using manual install:
    - Mac: Double-click the downloaded .dmg file and drag the MongoDB application to your Applications folder
    - Windows: Locate the downloaded file, which will be a .zip file. Right-click the zip file and select "Extract All" or use a program like 7-Zip or WinZip to extract the contents. Choose the directory (folder) where you want to extract the MongoDB files. On your Command Prompt or PowerShell, navigate to the MongoDB bin directory cd `C:\path\to\mongodb\bin` (replace “C:\path\to\mongodb” with with the actual path to your MongoDB installation directory) and run `mongod.exe` to start the server
    - Linux: Download the MongoDB tarball for Linux. Extract the contents to a directory of your choice. Navigate to the MongoDB bin directory and run `mongod` to start the server.

Getting Started

1. Open a terminal or command prompt on your system.
2. Run the following command to start the MongoDB server: `mongod`
    - By default, MongoDB will run and listen for connections on port 27017.
3. If you want to specify a different data directory, you can use the --dbpath option. For example: `mongod --dbpath /path/to/data/directory`

    * Replace “/path/to/data/directory” with the path to the directory where you want MongoDB to store its data.
4. The server will start and you will also see log messages indicating the server's status and configuration.

Tip: Keep the terminal window running while the MongoDB server is active. If you close the terminal, the server process will also terminate.

Using MongoDB Shell

1. Open a terminal or command prompt on your system.
2. To start the MongoDB shell, simply type the following command and press Enter: `mongo`
3. If your MongoDB server is running on the default port (27017) and is accessible on the local machine, the shell will connect to it.
4. You will see a welcome message and the MongoDB version information, along with the primary prompt (>). This indicates that you are now in the MongoDB shell.
5. You can start executing MongoDB commands and queries. For example, you can list databases, switch to a specific database, insert documents, perform queries, and more. Here are some commands you can use for these examples:
    - List the available databases: show dbs
    - Switch to a specific database or create a new one if it doesn't exist: `use` Example: `use mydatabase`
    - Insert a document into the current database's collection: `insertOne`
    - Perform queries: `find`

Here is an example of inserting a document into a collection in the MongoDB shell:

```
use mydatabase  // Switch to a specific database (replace "mydatabase" with your database name)
db.mycollection.insertOne({ name: "Max", age: 27 })
```

Databases and Collections

Databases: Think of a database in MongoDB like a virtual container for your data. It's similar to a folder on your computer where you can store related files. Each database can hold different types of information, just like how you have different folders for different types of files, such as documents, pictures, and music.

Collections: Inside each database, you have what are called collections. These are like sub-folders within your main folder. In each collection, you store specific types of information, just like you might have a "Documents" collection in your "Documents" folder and a "Photos" collection in your "Pictures" folder.

Basic Operations

Insert Documents: add documents (data) to a collection

- **insertOne:** `db.myCollection.insertOne({ name: "Max", age: 27 })`
- **insertMany:**

```
db.myCollection.insertMany([
    {
        name: "Max",
        age: 27
    },
    {
        name: "Alice",
        age: 30
    }
])
```

Query Documents: use the “find” method to retrieve documents from a collection.

- **Basic Find:**

```
db.myCollection.find()
```

- **Find w/ Specific Field Value:** retrieve documents where a specific field has a particular value

```
db.myCollection.find({ name: "Alice" })
```

- **Find with Multiple Conditions:** retrieve documents that match multiple conditions using logical operators

```
db.myCollection.find({ age: { $gte: 25, $lte: 30 }, gender: "female" })
```

  * In this operation, documents should have an "age" field with a value greater than or equal to 25 ($gte means "greater than or equal to") and less than or equal to 30 ($lte means "less than or equal to"). The “gender” should be a “female”

- **Projection (Selecting Fields):** choose specific fields to be included in the results and exclude others

```
db.myCollection.find({}, { name: 1, age: 1, _id: 0 })
```

  *

- **Sorting Results:** sort the query results based on a field in ascending (1) or descending (-1) order

```
db.myCollection.find().sort({ age: 1 }) // Ascending
db.myCollection.find().sort({ name: -1 }) // Descending
```

- **Limiting Results:** limit the number of documents returned by the query

```
// Limit to the first 5 documents
db.myCollection.find().limit(5)
```

- **Skipping Results:** skip a specified number of documents and return the rest

```
db.myCollection.find().skip(10) // Skip the first 10 documents
```

- **Query Operators:** utilize query operators like **$eq**, **$ne**, **$gt**, **$lt**, **$in**, **$regex**, and more for more specific queries

```
db.myCollection.find({ age: { $gt: 25 } }) // Find documents with age greater than 25
db.myCollection.find({ name: { $regex: /^A/ } }) // Find documents with names starting with 'A'
```

  - **$eq:** Matches exact values
  - **$ne:** Excludes specific values
  - **$gt:** Greater than
  - **$lt:** Less than
  - **$in:** Matches any in an array
  - **$regex:** Matches using regular expressions

- **Combining Operators:** You can combine various operations to create complex queries. For instance, you can filter by specific fields, sort the results, and limit the number of documents returned in a single query. Here is an example:

```
db.myCollection.find(
  {
    age: { $gte: 25, $lte: 30 }, // Age between 25 and 30
    gender: "female" // Gender is female
  },
  {
    name: 1, // Include the 'name' field
    age: 1, // Include the 'age' field
    _id: 0 // Exclude the default '_id' field
  }
).sort({ name: 1 }).limit(10)
```

BSON

- MongoDB stores data in a JSON-like format called BSON (Binary JSON). It is similar to the human-readable JSON format commonly used in web development.
- Documents in MongoDB are structured in a way that's easy to read and write for developers, just like JSON.
- One of the strengths of MongoDB is that it allows you to store documents with varying structures within the same collection. This flexibility is especially useful when working with evolving or unstructured data, as you don't need to adhere to a rigid, fixed schema.

Here is an example of an example collection named “Products”:
In a MongoDB collection named "Products," you might store data for various products

```
{
  "name": "Laptop",
  "brand": "Dell",
  "price": 999.99,
  "specs": {
    "CPU": "Intel Core i7",
    "RAM": "16GB",
    "Storage": "512GB SSD"
  }
},
{
  "name": "Smartphone",
  "brand": "Apple",
  "price": 799.99,
  "camera": "12MP",
  "OS": "iOS"
},
{
  "title": "Atomic Habits",
  "author": James Clear",
  "genre": "Self-Help",
  "publishedYear": 2018
},
{
  "name": "Android Smartphone",
  "brand": "Samsung",
  "price": 699.99,
  "camera": "16MP",
  "OS": "Android"
},
{
  "name": "MacBook Pro",
  "brand": "Apple",
  "price": 1499.99,
  "specs": {
    "CPU": "Intel Core i5",
    "RAM": "8GB",
    "Storage": "256GB SSD"
  }
}
```

There are 5 different documents in the “Products” collection: Two different laptops, two different smartphones, and a book

Document: a specific piece of data stored within a collection

Indexing

Indexing in MongoDB is a way to optimize data retrieval by creating a structured reference that speeds up search and sorting operations, much like an organized index in a book. It helps MongoDB find and deliver data efficiently, improving query performance.

Here are the types of indexes used in MongoDB:

- **Single-Field Indexes:** These indexes are created on a single field in a document. They significantly speed up queries that involve that specific field. For example, you might create a single-field index on the "username" field in a user collection to quickly find users by their usernames.

  ```
  db.users.createIndex({ username: 1 })
  ```

- **Compound Indexes:** Compound indexes are created on multiple fields in a document. They are beneficial when you need to query based on a combination of fields. For instance, you can create a compound index on "firstName" and "lastName" to efficiently query for users by their full names.

  ```
  db.users.createIndex({ firstName: 1, lastName: 1 })
  ```

- **Text Indexes:** Text indexes are designed for searching text content in documents. They allow you to perform full-text search, making it easier to find documents containing specific words or phrases. Text indexes are valuable for applications with extensive textual data, such as articles or comments.

  ```
  db.articles.createIndex({ content: "text" })
  ```

Aggregation

Aggregation allows you to perform advanced operations like filtering, grouping, sorting, and applying mathematical functions to the data to derive meaningful insights and reports. It is particularly useful for complex data analysis and reporting tasks.

The stages in a MongoDB aggregation pipeline need to be in a specific order. The order of the stages matters because each stage takes the output of the previous stage and performs further data transformation or filtering.

Here is the typical order:

- **First Stage:** Data from the collection (the starting point).
- **Intermediate Stages (e.g., $match, $group, $project, $sort, etc.):** These stages can be in any order depending on your data processing requirements.
- **Last Stage: The final stage (e.g., $project, $sort, $limit, etc.)** produces the output.

$match Stage (Filtering):

```
db.sales.aggregate([
  { $match: { date: { $gte: ISODate("2022-01-01") } } }
])
```

$group Stage (Grouping and Aggregating):

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

$project Stage (Field Projection and Renaming):

```
db.users.aggregate([
  { $project: { _id: 0, fullName: { $concat: ["$firstName", " ", "$lastName"] } } }
])
```

$sort Stage (Sorting):

```
db.products.aggregate([
  { $sort: { price: 1 } }
])
```

$limit Stage (Limiting Results):

```
db.customers.aggregate([
  { $limit: 10 }
])
```

$unwind Stage (Deconstructing Arrays):

```
db.orders.aggregate([
  { $unwind: "$items" }
])
```

$lookup Stage (Performing Joins):

```
db.orders.aggregate([
  {
    $lookup: {
      from: "products",
      localField: "productID",
      foreignField: "_id",
      as: "product"
    }
  }
])
```

Replication

In MongoDB, replication means making extra copies of your data on different servers. This is like having a backup. One server (we call it the main one) handles writing new data, while the others copy everything to be safe. If the main server has a problem, one of the backup servers can step in to keep your data safe and available. This is important to make sure your data is always there when you need it and won't get lost.

Example of replication in MongoDB (Social Media App):

Imagine you are running a social media app, and you want to ensure your users' data is always available, even if one server fails. You set up replication in the following way:

- **Primary Server (Server A):** This is your main server where all the new data is written. Users post photos and updates, and it goes to Server A.
- **Secondary Server 1 (Server B):** This is a copycat of Server A. It replicates (makes copies of) all the data from Server A.
- **Secondary Server 2 (Server C):** Another copycat server that also replicates everything from Server A.

If something happens to Server A, like a technical glitch or maintenance, Server B or Server C can step in and take over. Replication helps make sure your social media app is always up and running, even when there are technical issues with one of the servers. It's like having a backup that kicks in when you need it.

Sharding

Sharding is a technique that MongoDB uses to distribute and manage large data volumes across multiple servers, allowing the database to handle high throughput and massive data storage needs. MongoDB's support for automatic sharding simplifies the process of scaling your database horizontally to meet growing demands.

Example of sharding in MongoDB (eCommerce Website):

Suppose you run a popular eCommerce website, and the data about all your products has grown so large that it can't fit on a single server anymore. To handle this growth, you decide to use sharding. Here's how it works:

- **Sharding Key Selection:** You select a sharding key, which is a field used to distribute the data. In this case, you choose "product_category" as the sharding key because it makes sense to group products by categories.
- **Shard Servers:** You set up multiple servers, called "shards." Each shard will store a subset of your product data.
  - Shard 1: Responsible for electronics and gadgets.
  - Shard 2: Takes care of clothing and fashion products.
  - Shard 3: Handles books and media items.
- **Data Distribution:** MongoDB automatically distributes product data across the shards based on the "product_category." For example, all electronic products go to Shard 1, clothing products go to Shard 2, and so on.
- **Querying Data:** When a customer searches for electronics, the query goes to Shard 1. If someone looks for clothing items, the query is directed to Shard 2. MongoDB takes care of routing the queries to the appropriate shards based on the sharding key.
- **Balancing Data:** MongoDB also balances the data distribution across shards to ensure no single shard is overloaded. It can move data between shards as needed.

Security

MongoDB provides authentication and authorization mechanisms to protect your data. It is crucial to use these security features to ensure that only authorized users and applications can access and interact with your database, thus safeguarding your valuable information.

Example of security in MongoDB (User Authentication);

Suppose you're managing a company's employee database in MongoDB. To secure the data, you want to set up user authentication.

1. **Creating a User:**
   First, you create a user with specific credentials (username and password). This user will have access to the database.
   ```
   use admin
   db.createUser({
     user: "john_doe",
     pwd: "secure_password",
     roles: [{ role: "readWrite", db: "employee_data" }]
   })
   ```
2. **Enabling Authentication:**
   You ensure that authentication is enabled in MongoDB's configuration file (usually **mongod.conf**).
   ```
   security:
     authorization: enabled
   ```
3. **Access Control:**
   You set up access control, specifying which users have read or write access to the "employee_data" database.
   ```
   db.auth("john_doe", "secure_password")
   db.getCollection("employee_data").find()
   ```
4. **Secure Data Access:**
   Only users with the correct credentials (like "john_doe" with the right password) can access and interact with the "employee_data" database. Unauthorized users will be denied access.

The result of doing these steps will be that you ensure that only authorized individuals with the correct credentials can access and manage the employee data, enhancing the overall security of your MongoDB database.

Drivers

MongoDB provides official drivers for various programming languages, such as Python, Java, Node.js, and even more. These drivers make it easy to integrate MongoDB into your applications written in those languages, allowing you to interact with the database seamlessly.

Here are some examples of drivers in MongoDB:

PyMongo: The official MongoDB driver for Python, which allows Python developers to interact with MongoDB.

```python
from pymongo import MongoClient

# Connect to the MongoDB server
client = MongoClient('mongodb://localhost:27017/')

# Access a specific database and collection
db = client.mydatabase
collection = db.mycollection

# Insert a document
collection.insert_one({"name": "John Doe", "age": 30})

# Query documents
result = collection.find({"age": {"$gte": 25}})
```

MongoDB Java Driver: The official driver for Java applications, which allows Java developers to interact with MongoDB.

```java
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoCollection;
import org.bson.Document;

// Create a MongoDB client
MongoClient mongoClient = MongoClients.create("mongodb://localhost:27017");

// Access a collection
MongoCollection<Document> collection =
mongoClient.getDatabase("mydb").getCollection("mycollection");

// Insert a document
Document document = new Document("name", "Jane Smith").append("age", 28);
collection.insertOne(document);

// Query documents
List<Document> results = collection.find(eq("age", 30)).into(new ArrayList<>());
```

Node.js MongoDB Driver: The official MongoDB driver for Node.js (server-side JavaScript)

```
const { MongoClient } = require('mongodb');

// Create a MongoDB client
const client = new MongoClient('mongodb://localhost:27017');

// Connect to the MongoDB server
client.connect()
  .then(async () => {
    const db = client.db('mydb');
    const collection = db.collection('mycollection');

    // Insert a document
    await collection.insertOne({ name: 'Alice Johnson', age: 35 });

    // Query documents
    const results = await collection.find({ age: { $gte: 25 } }).toArray();
  })
  .catch(error => console.error(error))
  .finally(() => client.close());
```

Compass

Compass is a graphical user interface (GUI) tool that simplifies the management of MongoDB databases, collections, and documents. It provides an intuitive way to interact with MongoDB, making tasks like querying, data exploration, and administration more accessible for users.

Example of MongoDB Compass (Managing a Student Database):
Suppose you have a MongoDB database that stores student records. You want to use MongoDB Compass to perform various tasks.

1. **Connecting to the Database:**
    - Launch MongoDB Compass and connect to your MongoDB server by specifying the connection details (e.g., host, port, authentication credentials).
2. **Browsing Collections:**
    - Once connected, you can see a list of databases and collections in your database. Select the "students" collection.
3. **Viewing Documents:**
    - MongoDB Compass displays the documents in the "students" collection in a structured format. You can view student records, including their names, ages, and courses.
4. **Querying Data:**
    - Use the intuitive query builder to search for specific student records. For example, you can create a query to find all students who are enrolled in the "Computer Science" course.

5. **Adding New Documents:**
    - You can add new student records directly through the GUI by providing the student's information, such as name, age, and course.
6. **Modifying Documents:**
    - If a student's information changes, you can edit the corresponding document in the collection.
7. **Index Management:**
    - You can create and manage indexes on fields to improve query performance.
8. **Aggregation Pipeline:**
    - MongoDB Compass provides a visual aggregation pipeline builder, allowing you to create and execute complex data aggregation queries.
9. **Data Export:**
    - You can export data from the collection to various formats, such as JSON or CSV, for reporting or analysis.
10. **Server Stats:**
    - MongoDB Compass provides information about the server's performance and resource utilization.

Backups

Regularly back up your MongoDB data to prevent data loss. MongoDB provides tools like **mongodump**, **mongorestore**, File System Snapshots, and Third Party Backup Solutions for this purpose.

- **mongodump:**
    - A command-line tool provided by MongoDB, which allows you to create backups of your databases and collections by exporting the data in BSON format
    - You can specify which database or collection to back up

```
mongodump --db mydatabase --out /backup/
```

- **mongorestore:**
    - The counterpart to mongodump, which allows you to restore data from backup files created by **mongodump**
    - You specify the source directory containing the backup data

```
mongorestore --db mydatabase /backup/mydatabase/
```

- **File System Snapshots:**
    - Captures the database files at a point in time
    - Useful for consistent backups but may require specialized file system support (e.g., LVM snapshots or ZFS snapshots).
- **Third-Party Backup Solutions:**
    - Automates the backup process, provides point-in-time recovery, and offers redundancy (e.g., AWS Backup, Google Cloud Backup, MongoDB Atlas backup services, …)

Error Handling

Error handling in MongoDB involves the process of identifying, diagnosing, and addressing errors that may occur during database operations. MongoDB provides detailed error codes and messages to assist in this process.

- **Error Codes:** MongoDB assigns specific error codes to different types of errors. These error codes are numeric identifiers that help identify the nature of the issue.
- **Error Messages:** Along with error codes, MongoDB provides descriptive error messages that explain the cause of the error. These messages offer valuable information for troubleshooting.
- **Diagnosis:** When an error occurs, review the error code and message to diagnose the issue. Understanding the error's context and cause is essential for effective troubleshooting.
- **Troubleshooting:** Once the error is identified, take appropriate actions to resolve it. This may involve modifying queries, adjusting configurations, or handling exceptions in your application code.
- **Logging:** MongoDB provides options to log errors, allowing you to keep a record of issues for analysis and monitoring.