Introduction to No-SQL Databases ,
https://youtu.be/QKJwYVbDjWg

Introduction to NoSQL Databases

What are NoSQL Databases?

NoSQL databases, also known as "non-relational" or "not only SQL" databases, are designed for specific data models and have flexible schemas. They are built to handle large volumes of data, high user loads, and agile development processes. Unlike traditional relational databases (RDBMS), NoSQL databases do not use tables, rows, and columns to store data.

Key Characteristics of NoSQL Databases

1. Schema-less: NoSQL databases do not require a fixed schema, allowing for flexibility in data storage.
2. Scalability: Designed to scale out by distributing data across multiple servers.
3. High Performance: Optimized for specific data models and access patterns, providing faster read and write operations.
4. Distributed Computing: Built to run on clusters of computers, enhancing fault tolerance and data availability.
5. Horizontal Scaling: Can handle more traffic by adding more servers to the database.

Types of NoSQL Databases

1. Document Stores: Store data in JSON-like documents. Examples include MongoDB, CouchDB.
   - Key-Value Stores: Store data as key-value pairs. Examples include Redis, DynamoDB.
   - Column Stores: Store data in columns rather than rows. Examples include Cassandra, HBase.

- Graph Databases: Designed to store and navigate relationships. Examples include Neo4j, ArangoDB.

Advantages of NoSQL Databases

1. Flexibility: The ability to store structured, semi-structured, and unstructured data.
2. Scalability: Easy to scale out and handle large volumes of data and high traffic loads.
3. Performance: Optimized for specific use cases and data access patterns, often resulting in better performance.
4. Agility: Quick to develop and deploy applications due to schema-less nature.

Disadvantages of NoSQL Databases

1. Complexity: Managing and maintaining NoSQL databases can be complex, especially when scaling out.
2. Maturity: Some NoSQL databases are less mature than traditional RDBMS and may lack advanced features.
3. Standards: Lack of standardized query languages and interfaces.

Use Cases for NoSQL Databases

1. Big Data Applications: Handling large volumes of structured and unstructured data.
2. Real-Time Web Applications: High throughput and low latency requirements.
3. Content Management Systems: Flexible data models that evolve over time.
4. IoT and Sensor Data: Handling high-velocity data streams from various devices.

Key Concepts in NoSQL Databases

1. CAP Theorem: Describes the trade-offs between consistency, availability, and partition tolerance in distributed systems. NoSQL databases typically sacrifice consistency for availability and partition tolerance.

2. Eventual Consistency: In distributed systems, data will eventually become consistent, but immediate consistency is not guaranteed.

3. Sharding: Distributing data across multiple servers to ensure scalability and performance.

4. Replication: Copying data across multiple servers to ensure availability and fault tolerance.

Popular NoSQL Databases

1. MongoDB

- Type: Document Store

- Key Features: Flexible schema, high performance, horizontal scalability, rich query language.

- Use Cases: Content management, real-time analytics, mobile applications.

2. Cassandra

- Type: Column Store

- Key Features: High availability, fault tolerance, linear scalability, wide-column store.

- Use Cases: Time-series data, IoT, real-time data processing.

3. Redis

- Type: Key-Value Store

- Key Features: In-memory data storage, high performance, support for various data structures.

- Use Cases: Caching, real-time analytics, session storage.

4. Neo4j

- Type: Graph Database

- Key Features: Native graph storage, ACID transactions, powerful query language (Cypher).

- Use Cases: Social networks, recommendation engines, fraud detection.

Conclusion

NoSQL databases offer a variety of data storage solutions tailored to different application requirements. Their ability to handle large volumes of data, provide high performance, and support flexible data models makes them an essential component of modern data architectures. As a MERN stack professional, understanding the strengths and limitations of different NoSQL databases will enable you to choose the right tool for your application's needs.

Introduction to MongoDB,
https://youtu.be/wJqnOj2fo7o

Introduction to MongoDB

What is MongoDB?

MongoDB is a popular NoSQL database known for its high performance, scalability, and flexibility. It stores data in a JSONlike format called BSON (Binary JSON), making it easy to store complex data structures.

Key Features of MongoDB

1. DocumentOriented Storage: Stores data in flexible, JSONlike documents.
2. Schema Flexibility: Allows for dynamic schema, accommodating evolving data structures.
3. High Performance: Optimized for read and write operations.
4. Horizontal Scalability: Uses sharding to distribute data across multiple servers.
5. Rich Query Language: Supports adhoc queries, indexing, and realtime aggregation.
6. Replication: Ensures high availability with replica sets.
7. Aggregation Framework: Provides powerful data aggregation and transformation capabilities.

Core Concepts in MongoDB

1. Database: A container for collections.
2. Collection: A group of MongoDB documents, equivalent to a table in a relational database.
3. Document: A record in a collection, similar to a row in a table. Documents are stored in BSON format.
4. Indexing: Improves the performance of queries by creating indexes on fields.

5. Replication: Duplicates data across multiple servers for high availability.

6. Sharding: Distributes data across multiple servers to handle large datasets and high traffic.

Use Cases for MongoDB

1. Content Management Systems: Flexible schema accommodates various content types.

2. RealTime Analytics: High performance for realtime data processing.

3. Internet of Things (IoT): Handles highvelocity data streams.

4. Mobile Applications: Stores user data, session information, and application state.

Installation Steps for MongoDB

Installation on Windows

1. Download MongoDB Installer

Go to the [MongoDB Download Center](https://www.mongodb.com/try/download/community) and select the version for Windows.

Download the .msi installer.

2. Run the Installer

Doubleclick the downloaded .msi file.

Follow the installation prompts.

Choose "Complete" setup type for all features.

3. Configure Environment Variables

Add MongoDB to your system's PATH.

Open the System Properties > Environment Variables.

Edit the Path variable and add the path to the MongoDB bin directory (e.g., C:\Program Files\MongoDB\Server\<version>\bin).

4. Run MongoDB

Open Command Prompt.

Run mongod to start the MongoDB server.

Open another Command Prompt window.

Run mongo to start the MongoDB shell.

Installation on macOS

1. Install Homebrew

If Homebrew is not installed, install it using the following command:

sh

/bin/bash c "$(curl fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

2. Install MongoDB

Use Homebrew to install MongoDB:

sh

brew tap mongodb/brew

brew install mongodbcommunity

3. Start MongoDB

Start MongoDB as a service:

sh

brew services start mongodb/brew/mongodbcommunity

4. Run MongoDB Shell

Open Terminal.

Run mongo to start the MongoDB shell.

Installation on Linux

1. Import MongoDB Public Key

Import the MongoDB public GPG key:

sh

wget qO https://www.mongodb.org/static/pgp/server4.4.asc | sudo aptkey add

2. Create MongoDB List File

Create a list file for MongoDB:

sh

echo "deb [arch=amd64,arm64] https://repo.mongodb.org/apt/ubuntu focal/mongodborg/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodborg4.4.list

3. Update Package Database

Update the package database:

sh

sudo aptget update

4. Install MongoDB

Install MongoDB packages:

sh

sudo aptget install y mongodborg

5. Start MongoDB

Start the MongoDB service:

sh

sudo systemctl start mongod

6. Verify Installation

Check the status of MongoDB:

sh

sudo systemctl status mongod

7. Run MongoDB Shell

Open Terminal.

Run mongo to start the MongoDB shell.

MongoDB Compass Installation

MongoDB Compass is a graphical interface for MongoDB, providing an easy way to visualize and interact with your data.

1. Download MongoDB Compass

Go to the [MongoDB Compass Download Page](https://www.mongodb.com/try/download/compass).

Download the installer for your operating system.

2. Install MongoDB Compass

Run the installer and follow the installation prompts.

3. Connect to MongoDB

Open MongoDB Compass.

Enter the connection string for your MongoDB instance (default is mongodb://localhost:27017).

Click "Connect" to start using MongoDB Compass.

Conclusion

MongoDB is a powerful NoSQL database that provides flexibility, scalability, and high performance. Its documentoriented storage, rich query language, and robust features make it an excellent choice for a wide range of applications. Installing MongoDB is straightforward, with comprehensive steps for various operating systems. With MongoDB and tools like MongoDB Compass, you can efficiently manage and interact with your data.

Importing and Exporting Data in MongoDB, https://youtu.be/AIIdl053Hug

Importing and Exporting Data in MongoDB

Using MongoDB Compass

MongoDB Compass is a GUI tool that allows you to interact with your MongoDB databases visually. Here's how to import and export data using MongoDB Compass:

Importing Data with MongoDB Compass

1. Open MongoDB Compass:
   - Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection where you want to import the data.

3. Import Data:
   - Click on the "Add Data" button and select "Import File".
   - Choose the file type you want to import (JSON or CSV).
   - For JSON files:
     - Select the JSON file from your system.
     - Click "Open" and then "Import" to start the import process.
   - For CSV files:
     - Select the CSV file from your system.
     - Map the CSV fields to your MongoDB fields.
     - Click "Open" and then "Import" to start the import process.

Exporting Data with MongoDB Compass

1. Open MongoDB Compass:
   - Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection you want to export.

3. Export Data:
   - Click on the collection name.
   - Click the "Export Collection" button.
   - Select the export format (JSON or CSV).
   - Choose the fields you want to export (optional).
   - Click "Export" and specify the location to save the file on your system.

Using Mongo Shell

The Mongo shell is a command-line interface for interacting with MongoDB. It provides powerful tools for data import and export.

Importing Data with Mongo Shell

1. Prepare Your Data:
   - Ensure your data is in a compatible format, typically JSON or CSV.

2. Use `mongoimport`:
   - Open your terminal or command prompt.

- Use the `mongoimport` command to import data. Below are examples for JSON and CSV files:

For JSON Files:

```sh
mongoimport --uri mongodb://localhost:27017 --db yourDatabaseName --collection yourCollectionName --file /path/to/yourfile.json --jsonArray
```

For CSV Files:

```sh
mongoimport --uri mongodb://localhost:27017 --db yourDatabaseName --collection yourCollectionName --type csv --headerline --file /path/to/yourfile.csv
```

3. Parameters Explanation:

- `--uri`: Specifies the MongoDB connection string.
- `--db`: Specifies the database name.
- `--collection`: Specifies the collection name.
- `--file`: Specifies the path to the data file.
- `--jsonArray`: Indicates the file contains an array of JSON documents (for JSON files).
- `--type csv`: Indicates the file type is CSV (for CSV files).
- `--headerline`: Indicates the first line of the CSV file contains field names (for CSV files).

Exporting Data with Mongo Shell

1. Use `mongoexport`:

- Open your terminal or command prompt.

- Use the `mongoexport` command to export data. Below are examples for JSON and CSV files:

For JSON Files:

```sh
mongoexport --uri mongodb://localhost:27017 --db yourDatabaseName --collection yourCollectionName --out /path/to/outputfile.json --jsonArray
```

For CSV Files:

```sh
mongoexport --uri mongodb://localhost:27017 --db yourDatabaseName --collection yourCollectionName --type=csv --fields field1,field2,field3 --out /path/to/outputfile.csv
```

2. Parameters Explanation:

- `--uri`: Specifies the MongoDB connection string.

- `--db`: Specifies the database name.

- `--collection`: Specifies the collection name.

- `--out`: Specifies the path to the output file.

- `--jsonArray`: Exports the data as an array of JSON documents (for JSON files).

- `--type csv`: Indicates the file type is CSV (for CSV files).

- `--fields`: Specifies the fields to export (for CSV files).

Conclusion

Importing and exporting data in MongoDB is straightforward, whether you use MongoDB Compass for a graphical interface or Mongo Shell for command-line operations. MongoDB Compass provides an intuitive way to handle data without needing to remember command-line syntax, while Mongo Shell offers powerful and flexible commands for automation and scripting. Understanding both methods allows you to choose the best tool for your workflow and requirements.

CRUD Operations in MongoDB,
https://youtu.be/2b2LidhcutE

CRUD Operations in MongoDB

CRUD stands for Create, Read, Update, and Delete. These operations are fundamental to interacting with databases. Here, we'll cover how to perform these operations in MongoDB using both the MongoDB Compass GUI and the Mongo Shell.

Create Operation

The create operation involves inserting new documents into a collection.

Using MongoDB Compass

1. Open MongoDB Compass:
   - Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection where you want to insert data.

3. Insert Data:
   - Click on the "Insert Document" button.
   - Enter the JSON document in the provided editor.
   - Click "Insert" to add the document to the collection.

Using Mongo Shell

1. Open Mongo Shell:
   - Open your terminal or command prompt.

- Connect to your MongoDB instance by running the mongo command.

2. Insert Document:

- Use the insertOne or insertMany method to add documents to a collection.

Insert a Single Document:

javascript

```
db.collectionName.insertOne({ field1: "value1", field2: "value2" });
```

Insert Multiple Documents:

javascript

```
db.collectionName.insertMany([
  { field1: "value1", field2: "value2" },
  { field1: "value3", field2: "value4" }
]);
```

Read Operation

The read operation involves querying the database to retrieve documents.

Using MongoDB Compass

1. Open MongoDB Compass:

- Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection you want to query.

3. Query Data:
   - Use the query bar to input your query in JSON format.
   - Click "Find" to execute the query and view the results.

   Example Query:
   json
   { "field1": "value1" }

Using Mongo Shell

1. Open Mongo Shell:
   - Open your terminal or command prompt.
   - Connect to your MongoDB instance by running the mongo command.

2. Find Documents:
   - Use the find or findOne method to query documents in a collection.

   Find All Documents:
   javascript
   db.collectionName.find();

   Find Documents with a Condition:

javascript

```
db.collectionName.find({ field1: "value1" });
```

Find a Single Document:

javascript

```
db.collectionName.findOne({ field1: "value1" });
```

Update Operation

The update operation involves modifying existing documents in a collection.

Using MongoDB Compass

1. Open MongoDB Compass:
   - Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection you want to update.

3. Update Data:
   - Use the query bar to find the documents you want to update.
   - Click on the "Update" button and enter the update criteria and new values in JSON format.
   - Click "Update" to apply the changes.

Example Update:

json

```
{ "field1": "value1" },
{ $set: { "field2": "new value" } }
```

Using Mongo Shell

1. Open Mongo Shell:
   - Open your terminal or command prompt.
   - Connect to your MongoDB instance by running the mongo command.

2. Update Document:
   - Use the updateOne, updateMany, or replaceOne method to modify documents.

Update a Single Document:

javascript

```
db.collectionName.updateOne(
  { field1: "value1" },
  { $set: { field2: "new value" } }
);
```

Update Multiple Documents:

javascript

```
db.collectionName.updateMany(
  { field1: "value1" },
  { $set: { field2: "new value" } }
```

```
);
```

Replace a Document:

```
javascript
db.collectionName.replaceOne(
  { field1: "value1" },
  { field1: "value1", field2: "new value" }
);
```

Delete Operation

The delete operation involves removing documents from a collection.

Using MongoDB Compass

1. Open MongoDB Compass:
   - Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection you want to delete from.

3. Delete Data:
   - Use the query bar to find the documents you want to delete.
   - Click on the "Delete" button and confirm the deletion.

Example Delete:

json

```
{ "field1": "value1" }
```

Using Mongo Shell

1. Open Mongo Shell:
   - Open your terminal or command prompt.
   - Connect to your MongoDB instance by running the mongo command.

2. Delete Document:
   - Use the deleteOne or deleteMany method to remove documents.

Delete a Single Document:

javascript

```
db.collectionName.deleteOne({ field1: "value1" });
```

Delete Multiple Documents:

javascript

```
db.collectionName.deleteMany({ field1: "value1" });
```

Conclusion

CRUD operations are essential for managing data in MongoDB. MongoDB Compass provides an intuitive GUI for these operations, while Mongo Shell

offers powerful command-line capabilities for more advanced users. Understanding both methods enables you to efficiently perform CRUD operations, whether you're developing applications or managing databases.

Updating and Deleting Documents in MongoDB, https://youtu.be/jBECp70Xqbk

Update Operations in MongoDB

MongoDB provides several methods to update documents in a collection. The most commonly used methods are updateOne, updateMany, and replaceOne.

1. updateOne

The updateOne method updates a single document that matches the specified filter. If multiple documents match the filter, only the first matching document is updated.

Syntax:

```
db.collection.updateOne(
   <filter>,
   <update>,
   <options>
)
```

filter: Specifies the selection criteria for the update. This is a query that matches documents.

update: Specifies the modifications to apply.

options: Optional. Additional options that affect the update operation.

Example:

Update the price of a product named "Laptop" to 1000.

```
db.products.updateOne(
```

```
  { name: "Laptop" },
  { $set: { price: 1000 } }
)
```

2. updateMany

The updateMany method updates all documents that match the specified filter.

Syntax:

```
db.collection.updateMany(
  <filter>,
  <update>,
  <options>
)
```

Example:
Update the inStock status to false for all products in the "Electronics" category.

```
db.products.updateMany(
  { category: "Electronics" },
  { $set: { inStock: false } }
)
```

3. replaceOne

The replaceOne method replaces a single document that matches the specified filter with a new document.

Syntax:

```
db.collection.replaceOne(
  <filter>,
  <replacement>,
  <options>
)
```

filter: Specifies the selection criteria for the update.

replacement: The new document that will replace the existing document.

Example:

Replace the document where the product name is "Desk" with a new document.

```
db.products.replaceOne(
  { name: "Desk" },
  { name: "Desk", category: "Office Furniture", price: 250, inStock: true }
)
```

Update Operators

$set: Sets the value of a field.

$inc: Increments the value of a field by a specified amount.

$mul: Multiplies the value of a field by a specified amount.

$rename: Renames a field.

$unset: Removes a field from a document.

Example:

Increment the price of a product named "Laptop" by 200.

```
db.products.updateOne(
  { name: "Laptop" },
  { $inc: { price: 200 } }
)
```

Delete Operations in MongoDB

MongoDB provides two methods for deleting documents from a collection: deleteOne and deleteMany.

1. deleteOne

The deleteOne method deletes a single document that matches the specified filter. If multiple documents match the filter, only the first matching document is deleted.

Syntax:

```
db.collection.deleteOne(
  <filter>,
  <options>
)
```

filter: Specifies the selection criteria for the deletion.
options: Optional. Additional options that affect the deletion operation.

Example:
Delete the product named "Laptop".

```
db.products.deleteOne({ name: "Laptop" })
```

2. deleteMany

The deleteMany method deletes all documents that match the specified filter.

Syntax:

```
db.collection.deleteMany(
  <filter>,
  <options>
)
```

Example:

Delete all products in the "Electronics" category.

```
db.products.deleteMany({ category: "Electronics" })
```

Example Data Setup

Before performing these operations, let’s assume we have the following products collection:

```
json
[
  { "_id": 1, "name": "Laptop", "category": "Electronics", "price": 1200, "inStock": true },
  { "_id": 2, "name": "Mouse", "category": "Electronics", "price": 25, "inStock": true },
  { "_id": 3, "name": "Keyboard", "category": "Electronics", "price": 75, "inStock": true },
  { "_id": 4, "name": "Chair", "category": "Furniture", "price": 150, "inStock": false },
  { "_id": 5, "name": "Desk", "category": "Furniture", "price": 300, "inStock": true }
]
```

Practical Use Cases

Update Use Case:

1. Discount on Electronics:
   Apply a 10% discount on all electronics.

```
db.products.updateMany(
  { category: "Electronics" },
  { $mul: { price: 0.9 } }
)
```

2. Restock Furniture:
   Set the inStock status to true for all furniture items.

```
db.products.updateMany(
  { category: "Furniture" },
  { $set: { inStock: true } }
)
```

Delete Use Case:

1. Remove OutofStock Products:
   Delete all products that are not in stock.

```
db.products.deleteMany({ inStock: false })
```

2. Clear Specific Category:

Delete all products in the "Furniture" category.

```
db.products.deleteMany({ category: "Furniture" })
```

By mastering these update and delete operations, you can effectively manage and manipulate your data in MongoDB, ensuring your database remains accurate and uptodate.

Aggregation Pipeline in MongoDB(Part 1),
https://youtu.be/51jf_zVILGg

Introduction to Aggregation

Aggregation operations process data records and return computed results. They are similar to SQL's GROUP BY and HAVING clauses. The MongoDB aggregation pipeline consists of stages in which each stage performs an operation on the input data and passes the result to the next stage.

The stages we'll cover are:

1. $match: Filters documents.
2. $group: Groups documents by a specified field and performs aggregation operations.
3. $project: Reshapes documents by including, excluding, or adding new fields.

1. $match Stage

The $match stage filters documents to pass only those that match the specified condition(s) to the next stage in the pipeline. It is similar to the WHERE clause in SQL.

Syntax:

{ $match: { <query> } }

Example 1: Filter by Category

Filter documents to include only those in the "Electronics" category.

```
db.products.aggregate([
  { $match: { category: "Electronics" } }
])
```

Example 2: Filter by Price Range

Filter documents where the price is greater than 100 and less than 1000.

```
db.products.aggregate([
  { $match: { price: { $gt: 100, $lt: 1000 } } }
])
```

Example 3: Filter by Stock Status

Filter documents to include only those that are in stock.

```
db.products.aggregate([
  { $match: { inStock: true } }
])
```

2. $group Stage

The $group stage groups input documents by the specified _id expression and applies the accumulator expressions to each group.

Syntax:

{ $group: { _id: <expression>, <field1>: { <accumulator1>: <expression1> }, ... } }

_id: The group key. Can be a single field, a compound key, or computed from an expression.

accumulator: Operations such as $sum, $avg, $min, $max, $push, $addToSet, etc.

Example 1: Group by Category and Sum Prices

Group documents by the category field and calculate the total price for each category.

```
db.products.aggregate([
  { $group: { _id: "$category", totalPrice: { $sum: "$price" } } }
])
```

Example 2: Group by Category and Average Price

Group documents by the category field and calculate the average price for each category.

```
db.products.aggregate([
  { $group: { _id: "$category", averagePrice: { $avg: "$price" } } }
```

])

Example 3: Group by Stock Status and Count Products

Group documents by the inStock field and count the number of products in each group.

```
db.products.aggregate([
  { $group: { _id: "$inStock", count: { $sum: 1 } } }
])
```

3. $project Stage

The $project stage reshapes each document in the stream by adding, removing, or renaming fields. It can also create new fields by using expressions.

Syntax:

```
{ $project: { <field1>: <expression1>, <field2>: <expression2>, ... } }
```

<field>: The field name.

<expression>: Specifies the value of the field, can be 1 (include), 0 (exclude), or an expression.

Example 1: Include Specific Fields

Include only the name and price fields in the output documents.

```
db.products.aggregate([
  { $project: { name: 1, price: 1, _id: 0 } }
])
```

Example 2: Add a Computed Field

Add a new field totalValue that is the product of price and quantity.

```
db.products.aggregate([
  { $project: { name: 1, totalValue: { $multiply: ["$price", "$quantity"] } } }
])
```

Example 3: Rename Fields

Rename the price field to cost.

```
db.products.aggregate([
  { $project: { name: 1, cost: "$price", _id: 0 } }
])
```

Combined Example: Using $match, $group, and $project

Let’s combine these stages into a single pipeline to perform a complex aggregation.

Scenario: Find the total sales value of instock electronics products, grouped by product name.

Data Setup:

Assume we have the following products collection:

```
[
  { "_id": 1, "name": "Laptop", "category": "Electronics", "price": 1200, "quantity": 10, "inStock": true },
  { "_id": 2, "name": "Mouse", "category": "Electronics", "price": 25, "quantity": 200, "inStock": true },
  { "_id": 3, "name": "Keyboard", "category": "Electronics", "price": 75, "quantity": 100, "inStock": true },
  { "_id": 4, "name": "Chair", "category": "Furniture", "price": 150, "quantity": 50, "inStock": false },
  { "_id": 5, "name": "Desk", "category": "Furniture", "price": 300, "quantity": 20, "inStock": true }
]
```

Pipeline:

```
db.products.aggregate([
  { $match: { category: "Electronics", inStock: true } },
  { $group: { _id: "$name", totalSales: { $sum: { $multiply: ["$price", "$quantity"] } } } },
```

```
  { $project: { productName: "$_id", totalSales: 1, _id: 0 } }
])
```

Explanation:

1. $match: Filters the documents to include only those in the "Electronics" category and that are in stock.

2. $group: Groups the filtered documents by product name and calculates the total sales value (price quantity) for each product.

3. $project: Renames the _id field to productName and includes the totalSales field in the output documents, excluding the original _id.

Result:

```
[
  { "productName": "Laptop", "totalSales": 12000 },
  { "productName": "Mouse", "totalSales": 5000 },
  { "productName": "Keyboard", "totalSales": 7500 }
]
```

Aggregation Pipeline in MongoDB(Part 2),
https://youtu.be/o7HgxfarEQo

Aggregation Stages in MongoDB: Limit, Sort, and Count

1. Limit Stage ($limit)

The $limit stage restricts the number of documents that pass through the pipeline to the specified number.

Syntax:

```
{
  $limit: <positive integer>
}
```

Example:

```
js
db.collection.aggregate([
  { $limit: 5 }
]);
```

Use Case:

Useful for paginating results or for cases where only a subset of the data is needed.

2. Sort Stage ($sort)

The $sort stage orders the documents based on one or more fields. The sorting order can be ascending (1) or descending (1).

Syntax:

```
{
  $sort: { <field1>: <sort order>, <field2>: <sort order>, ... }
}
```

Example:

```
js
db.collection.aggregate([
  { $sort: { age: 1, name: 1 } }
]);
```

This will sort the documents first by the age field in ascending order, and then by the name field in descending order for documents with the same age.

Use Case:

Sorting is essential for presenting data in a userfriendly manner, such as listing users by their age or sorting products by price.

3. Count Stage ($count)

The $count stage returns the count of documents in the aggregation pipeline. The result is a single document with a count field.

Syntax:

```
{
  $count: <string>
}
```

Example:

```
js
db.collection.aggregate([
  { $match: { status: "active" } },
  { $count: "activeCount" }
]);
```

This will count the number of documents with the status field equal to "active" and return a document like:

```
{ "activeCount": <number> }
```

Use Case:

Counting is useful for generating reports, such as counting the number of active users or the total number of orders.

Detailed Example Combining Limit, Sort, and Count

Scenario

Suppose you have a collection of users and you want to find the top 10 oldest users, sort them by their registrationDate, and count how many users are in the database.

Example Aggregation Pipeline

js

```
db.users.aggregate([
  { $sort: { age: 1 } },              // Sort users by age in descending order (oldest first)
  { $limit: 10 },                     // Limit the result to the top 10 users
  { $sort: { registrationDate: 1 } },  // Sort these top 10 users by registration date in ascending order
  { $count: "totalTopUsers" }          // Count the number of documents in this pipeline (should be 10)
]);
```

Explanation

1. Sort: First, sort all users by age in descending order.

2. Limit: Limit the result set to the top 10 oldest users.

3. Sort: Resort these top 10 users by their registration date in ascending order.

4. Count: Finally, count how many documents are in the pipeline, which will be 10 in this case.

Practical Considerations

Performance: Sorting operations can be resourceintensive, especially on large datasets. Ensure proper indexing to optimize performance.

Memory Usage: The $sort stage requires sufficient memory to hold the documents being sorted. If the pipeline exceeds the memory limit, consider using the allowDiskUse option.

Pipeline Order: The order of stages in the pipeline matters. For instance, placing a $limit stage before a $sort can reduce the number of documents that need to be sorted, improving performance.

Final Thoughts

Understanding and effectively using aggregation stages like $limit, $sort, and $count is crucial for performing complex queries and data manipulations in MongoDB. These stages help streamline the data processing workflow, making it easier to derive meaningful insights from your data.

Aggregation Pipeline in MongoDB(Part 3),
https://youtu.be/wpYDocsZiCI

Aggregation Stages in MongoDB: Add Fields, Count, and Lookup

1. Add Fields Stage ($addFields)

The $addFields stage adds new fields to documents or modifies existing fields. The new field can be derived from existing fields or static values.

Syntax:

json

```
{
  $addFields: {
    <newField>: <expression>,
    ...
  }
}
```

Example:

js

```
db.collection.aggregate([
  { $addFields: { fullName: { $concat: ["$firstName", " ", "$lastName"] } } }
]);
```

This example concatenates firstName and lastName fields to create a new fullName field.

Use Case:

Adding or modifying fields is useful for transforming documents to meet specific requirements without changing the original documents in the collection.

2. Count Stage ($count)

The $count stage returns the count of documents in the aggregation pipeline. The result is a single document with a specified field name for the count.

Syntax:

json

```
{
  $count: <string>
}
```

Example:

js

```
db.collection.aggregate([
  { $match: { status: "active" } },
  { $count: "activeCount" }
]);
```

This will count the number of documents with the status field equal to "active" and return a document like:

json

```
{ "activeCount": <number> }
```

Use Case:

Counting is useful for generating reports, such as counting the number of active users or the total number of orders.

3. Lookup Stage ($lookup)

The $lookup stage performs a left outer join to a collection in the same database to filter in documents from the "joined" collection for processing.

Syntax:

```
json
{
  $lookup: {
    from: <collection>,
    localField: <field>,
    foreignField: <field>,
    as: <output array field>
  }
}
```

Example:

```
js
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
```

```
      localField: "customerId",
      foreignField: "_id",
      as: "customerDetails"
    }
  }
]);
```

This example joins the orders collection with the customers collection where the customerId in orders matches the _id in customers. The joined data is stored in the customerDetails field.

Use Case:

Joining data from multiple collections is essential for creating comprehensive reports and performing complex queries involving related data.

Detailed Example Combining Add Fields, Count, and Lookup

Scenario

Suppose you have an orders collection and a customers collection. You want to:

1. Add a field to calculate the total order value by multiplying quantity and price.

2. Count the number of orders placed by each customer.

3. Include customer details in each order.

Example Aggregation Pipeline

js

```
db.orders.aggregate([
  // Add a field to calculate the total order value
```

```
  {
    $addFields: {
      totalValue: { $multiply: ["$quantity", "$price"] }
    }
  },
  // Lookup to join with customers collection
  {
    $lookup: {
      from: "customers",
      localField: "customerId",
      foreignField: "_id",
      as: "customerDetails"
    }
  },
  // Group by customerId to count orders per customer
  {
    $group: {
      _id: "$customerId",
      totalOrders: { $count: {} },
      totalOrderValue: { $sum: "$totalValue" }
    }
  }
]);
```

Explanation

1. Add Fields: Adds a new field totalValue by multiplying quantity and price for each order.

2. Lookup: Joins the orders collection with the customers collection to include customer details.

3. Group: Groups the documents by customerId, counting the number of orders and summing the total order value for each customer.

Practical Considerations

Performance: The $lookup stage can be resourceintensive, especially with large collections. Ensure indexes on the localField and foreignField for optimal performance.

Memory Usage: The $addFields and $group stages may require sufficient memory for processing complex calculations and aggregations.

Pipeline Order: Proper ordering of stages is crucial for efficient data processing. For instance, filtering data before performing a lookup can reduce the amount of data processed, improving performance.

Final Thoughts

Understanding and effectively using aggregation stages like $addFields, $count, and $lookup is crucial for performing complex queries and data manipulations in MongoDB. These stages help streamline the data processing workflow, making it easier to derive meaningful insights from your data.

Lab program 6(MongoDB),
https://youtu.be/72oAa_O5xpY

CRUD Operations in MongoDB

CRUD stands for Create, Read, Update, and Delete. These operations are fundamental to interacting with databases. Here, we'll cover how to perform these operations in MongoDB using both the MongoDB Compass GUI and the Mongo Shell.

Create Operation

The create operation involves inserting new documents into a collection.

Using MongoDB Compass

1. Open MongoDB Compass:
   - Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection where you want to insert data.

3. Insert Data:
   - Click on the "Insert Document" button.
   - Enter the JSON document in the provided editor.
   - Click "Insert" to add the document to the collection.

Using Mongo Shell

1. Open Mongo Shell:
   - Open your terminal or command prompt.

- Connect to your MongoDB instance by running the mongo command.

2. Insert Document:

- Use the insertOne or insertMany method to add documents to a collection.

Insert a Single Document:

javascript

```
db.collectionName.insertOne({ field1: "value1", field2: "value2" });
```

Insert Multiple Documents:

javascript

```
db.collectionName.insertMany([
  { field1: "value1", field2: "value2" },
  { field1: "value3", field2: "value4" }
]);
```

Read Operation

The read operation involves querying the database to retrieve documents.

Using MongoDB Compass

1. Open MongoDB Compass:

- Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection you want to query.

3. Query Data:
   - Use the query bar to input your query in JSON format.
   - Click "Find" to execute the query and view the results.

   Example Query:
   json
   { "field1": "value1" }

Using Mongo Shell

1. Open Mongo Shell:
   - Open your terminal or command prompt.
   - Connect to your MongoDB instance by running the mongo command.

2. Find Documents:
   - Use the find or findOne method to query documents in a collection.

   Find All Documents:
   javascript
   db.collectionName.find();

   Find Documents with a Condition:

javascript

```
db.collectionName.find({ field1: "value1" });
```

Find a Single Document:

javascript

```
db.collectionName.findOne({ field1: "value1" });
```

Update Operation

The update operation involves modifying existing documents in a collection.

Using MongoDB Compass

1. Open MongoDB Compass:
   - Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection you want to update.

3. Update Data:
   - Use the query bar to find the documents you want to update.
   - Click on the "Update" button and enter the update criteria and new values in JSON format.
   - Click "Update" to apply the changes.

Example Update:

json

```
{ "field1": "value1" },
{ $set: { "field2": "new value" } }
```

Using Mongo Shell

1. Open Mongo Shell:
   - Open your terminal or command prompt.
   - Connect to your MongoDB instance by running the mongo command.

2. Update Document:
   - Use the updateOne, updateMany, or replaceOne method to modify documents.

Update a Single Document:

javascript

```
db.collectionName.updateOne(
  { field1: "value1" },
  { $set: { field2: "new value" } }
);
```

Update Multiple Documents:

javascript

```
db.collectionName.updateMany(
  { field1: "value1" },
  { $set: { field2: "new value" } }
```

```
);
```

Replace a Document:

```
javascript
db.collectionName.replaceOne(
  { field1: "value1" },
  { field1: "value1", field2: "new value" }
);
```

Delete Operation

The delete operation involves removing documents from a collection.

Using MongoDB Compass

1. Open MongoDB Compass:
   - Start MongoDB Compass and connect to your MongoDB instance.

2. Select the Database and Collection:
   - Choose the database and the specific collection you want to delete from.

3. Delete Data:
   - Use the query bar to find the documents you want to delete.
   - Click on the "Delete" button and confirm the deletion.

Example Delete:

json

```
{ "field1": "value1" }
```

Using Mongo Shell

1. Open Mongo Shell:
   - Open your terminal or command prompt.
   - Connect to your MongoDB instance by running the mongo command.

2. Delete Document:
   - Use the deleteOne or deleteMany method to remove documents.

Delete a Single Document:

javascript

```
db.collectionName.deleteOne({ field1: "value1" });
```

Delete Multiple Documents:

javascript

```
db.collectionName.deleteMany({ field1: "value1" });
```

Conclusion

CRUD operations are essential for managing data in MongoDB. MongoDB Compass provides an intuitive GUI for these operations, while Mongo Shell

offers powerful command-line capabilities for more advanced users. Understanding both methods enables you to efficiently perform CRUD operations, whether you're developing applications or managing databases.

Integration with Express,
https://youtu.be/V9x5vuGuuqU

Integrating MongoDB with Express involves several steps and concepts. These notes cover the detailed theory and key points for each step in the integration process.

1. Overview of MongoDB and Express

MongoDB

NoSQL Database: MongoDB is a NoSQL, documentoriented database. It stores data in flexible, JSONlike documents.

Collections and Documents: Data is organized into collections and documents. Collections are analogous to tables in relational databases, and documents are analogous to rows.

Schemaless: MongoDB does not require a predefined schema, allowing for flexible data models.

Express

Web Framework for Node.js: Express is a minimal and flexible Node.js web application framework that provides robust features for web and mobile applications.

Middleware: Express uses middleware functions to handle requests, responses, and the next middleware function in the application’s requestresponse cycle.

2. Setting Up the Environment

Prerequisites

MongoDB Installation: Install MongoDB and ensure it is running on your machine or server.

Node.js and npm: Install Node.js and npm (Node Package Manager).

Installing Required Packages

Express and Mongoose: Install the necessary packages using npm.

sh

```
npm install express mongoose
```

3. Connecting to MongoDB

Using Mongoose

Mongoose: An Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.

```
const express = require('express');
const mongoose = require('mongoose');

const app = express();
const port = 3000;

mongoose.connect('mongodb://localhost:27017/mydatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => {
  console.log('Connected to MongoDB');
}).catch((err) => {
  console.error('Error connecting to MongoDB:', err);
});
```

```
app.listen(port, () => {
  console.log(Server is running on port ${port});
});
```

4. Defining Schemas and Models

Mongoose Schema and Model

Schema: A Mongoose schema defines the structure of the document, default values, validators, etc.

Model: A model is a compiled version of the schema and is responsible for creating and reading documents from the underlying MongoDB database.

```
const { Schema, model } = mongoose;

const userSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
```

```
    type: String,
    required: true,
  },
});

const User = model('User', userSchema);
```

5. Creating Routes for CRUD Operations

Express Routes

CRUD Operations: Create, Read, Update, and Delete operations are essential for managing data. Express routes handle these operations by mapping HTTP requests to specific paths and methods.

```
app.use(express.json());

app.post('/users', async (req, res) => {
  try {
    const user = new User(req.body);
    await user.save();
    res.status(201).send(user);
  } catch (error) {
    res.status(400).send(error);
  }
});
```

```
app.get('/users', async (req, res) => {
  try {
    const users = await User.find();
    res.status(200).send(users);
  } catch (error) {
    res.status(500).send(error);
  }
});

app.get('/users/:id', async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) {
      return res.status(404).send();
    }
    res.status(200).send(user);
  } catch (error) {
    res.status(500).send(error);
  }
});

app.put('/users/:id', async (req, res) => {
  try {
    const user = await User.findByIdAndUpdate(req.params.id, req.body, { new:
true, runValidators: true });
    if (!user) {
      return res.status(404).send();
    }
```

```
    res.status(200).send(user);
  } catch (error) {
    res.status(400).send(error);
  }
});

app.delete('/users/:id', async (req, res) => {
  try {
    const user = await User.findByIdAndDelete(req.params.id);
    if (!user) {
      return res.status(404).send();
    }
    res.status(200).send(user);
  } catch (error) {
    res.status(500).send(error);
  }
});
```

6. Handling Errors

Middleware for Error Handling

Error Handling: Use Express middleware to catch and handle errors gracefully.

```
app.use((err, req, res, next) => {
  console.error(err.stack);
```

```
  res.status(500).send('Something broke!');
});
```

Summary

Integrating MongoDB with Express involves setting up the environment, connecting to MongoDB using Mongoose, defining schemas and models, creating routes for CRUD operations, handling errors, following security best practices, optimizing performance, testing, and documenting your API. Following these steps ensures a robust and scalable application.

Lab Program 7(Integration of Mongo & Express),
https://youtu.be/4-XIyCS-ZTQ

```
npm init -y
npm install mongoose
npm install body-parser
npm install express
```

server.js

```
const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');

// Initialize Express app
const app = express();
const port = 3000;

// Middleware
app.use(bodyParser.urlencoded({ extended: true }));

// Connect to MongoDB
mongoose.connect('mongodb://localhost:27017/studentDB', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

// Define a student schema
const studentSchema = new mongoose.Schema({
```

```
  rollNo: Number,
  name: String,
  age: Number,
  class: String
});

// Create a model
const Student = mongoose.model('Student', studentSchema);

// Serve the HTML form
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

// Handle form submission
app.post('/getStudent', async (req, res) => {
  const rollNo = req.body.rollNo;

  try {
    const student = await Student.findOne({ rollNo: rollNo }).exec();

    if (!student) {
      res.send('No student found with the given roll number.');
    } else {
      res.send(`
        <h1>Student Details</h1>
        <p>Roll No: ${student.rollNo}</p>
```

```
        <p>Name: ${student.name}</p>
        <p>Age: ${student.age}</p>
        <p>Class: ${student.class}</p>
        <a href="/">Search another student</a>
      `);
    }
  } catch (err) {
    res.send('Error occurred while fetching student details.');
  }
});

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Fetch Student Details</title>
</head>
<body>
  <h1>Fetch Student Details</h1>
  <form action="/getStudent" method="POST">
    <label for="rollNo">Roll No:</label>
```

```
    <input type="text" id="rollNo" name="rollNo" required>
    <button type="submit">Get Details</button>
  </form>
</body>
</html>
```

LabProgram-1(Javascript),
https://youtu.be/4lrfeoXwnu0

```
// 1.a to swap the case of all the characters in the given string
function swapCase(str) {
    return str.split('').map(char => {
      if (char === char.toUpperCase()) {
        return char.toLowerCase();
      } else {
        return char.toUpperCase();
      }
    }).join('');
  }

  const input = "The Quick Brown Fox";
  console.log(swapCase(input));

/*1 b.to find the most frequently occuring element in an array*/
function mostFrequent(arr) {
    const frequency = {};
    let max = 0;
    let result = null;

    arr.forEach(item => {
      frequency[item] = (frequency[item] || 0) + 1;
      if (frequency[item] > max) {
        max = frequency[item];
        result = item;
      }
    });
```

```
  return result;
}

const arr = [2, 2, 2, 1, 2, 1];
console.log(mostFrequent(arr));



//1 c.to remove duplicate value from the given array(ignore case sensitivity)
function removeDuplicates(arr) {
  return [...new Set(arr.map(item => item.toLowerCase()))];
}

const arr = ['a', 'b', 'A', 'B', 'a','c'];
console.log(removeDuplicates(arr));

//1d. Perform Binary search
function binarySearch(arr, target) {
  let start = 0;
  let end = arr.length - 1;

  while (start <= end) {
    const mid = Math.floor((start + end) / 2);

    if (arr[mid] === target) {
      return mid;
    } else if (arr[mid] < target) {
```

```
      start = mid + 1;
    } else {
      end = mid - 1;
    }
  }

  return -1;
}

const sortedArr = [1, 2, 3, 4, 5];
console.log(binarySearch(sortedArr, 3));

//1 e.to print the properties of a given object
const obj = {
  name: 'John',
  age: 30,
  profession: 'Developer',
  address:'Germany'
};

console.log(Object.keys(obj));


//1f.To check wether the given object has the given property
const obj = {
  name: 'John',
  age: 30,
```

```
  profession: 'Developer'
};

function hasProperty(obj, prop) {
  return obj.hasOwnProperty(prop);
}

console.log(hasProperty(obj, 'age')); // true
console.log(hasProperty(obj, 'salary')); // false



//1g. To perform Quicksort
function quickSort(arr) {
  if (arr.length <= 1) {
    return arr;
  }

  const pivot = arr[Math.floor(arr.length / 2)];
  const left = arr.filter(x => x < pivot);
  const middle = arr.filter(x => x === pivot);
  const right = arr.filter(x => x > pivot);

  return [...quickSort(left), ...middle, ...quickSort(right)];
}

const unsortedArr = [3, 6, 8, 10, 1, 2, 1];
```

```
  console.log(quickSort(unsortedArr));



//1h. to implement the bubblesort
function bubbleSort(arr) {
    const n = arr.length;
    for (let i = 0; i < n; i++) {
      for (let j = 0; j < n - 1; j++) {
        if (arr[j] > arr[j + 1]) {
          [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
        }
      }
    }
    return arr;
  }

  const unsortedArr = [64, 34, 25, 12, 22, 11, 90];
  console.log(bubbleSort(unsortedArr));



<!--1i.to read from a json object and print the values on an html form-->
<!DOCTYPE html>
<html>
<head>
  <title>JSON Data Table</title>
</head>
<body>
```

```
<table id="data-table" border="1">
  <thead>
    <tr>
      <th>Name</th>
      <th>Age</th>
      <th>Profession</th>
    </tr>
  </thead>
  <tbody>
  </tbody>
</table>

<script>
  const data = [
    { "name": "John", "age": 30, "profession": "Developer" },
    { "name": "Jane", "age": 25, "profession": "Designer" },
    { "name": "Mike", "age": 32, "profession": "Manager" }
  ];

  const tableBody = document.getElementById('data-
table').getElementsByTagName('tbody')[0];
  data.forEach(item => {
    const row = tableBody.insertRow();
    Object.values(item).forEach(value => {
      const cell = row.insertCell();
      cell.textContent = value;
    });
  });
```

```
  </script>
</body>
</html>


<!--1j.takes name,rollno,marks from a form and on form submissiondisplays them in a tabular format with GPA(like mark sheet)-->
<!DOCTYPE html>
<html>
<head>
  <title>Student Marks</title>
</head>
<body>
  <form id="student-form">
    <label>Name: <input type="text" id="name" required></label><br>
    <label>Roll No: <input type="text" id="rollno" required></label><br>
    <label>Marks: <input type="number" id="marks" required></label><br>
    <button type="submit">Submit</button>
  </form>

  <table id="marks-table" border="1">
    <thead>
      <tr>
        <th>Name</th>
        <th>Roll No</th>
        <th>Marks</th>
        <th>GPA</th>
      </tr>
    </thead>
```

```
    <tbody>
    </tbody>
  </table>

  <script>
    document.getElementById('student-form').addEventListener('submit',
function(event) {
      event.preventDefault();
      const name = document.getElementById('name').value;
      const rollno = document.getElementById('rollno').value;
      const marks = document.getElementById('marks').value;
      const gpa = (marks / 10).toFixed(2);

      const tableBody = document.getElementById('marks-
table').getElementsByTagName('tbody')[0];
      const row = tableBody.insertRow();
      [name, rollno, marks, gpa].forEach(value => {
        const cell = row.insertCell();
        cell.textContent = value;
      });
    });
  </script>
</body>
</html>
```