Lecture 14 - MongoDB with NodeJS II

Updating with $pull operator

The $pull operator in MongoDB removes elements from an array that match a specified condition. It's often used to remove specific values from arrays within documents.

Example:

```
db.collection.updateOne(
  { _id: ObjectId("your_document_id") },
  { $pull: { arrayField: { $in: ["value1", "value2"] } } }
);
```

In this example, elements matching "value1" and "value2" will be removed from the arrayField.

Indexes in MongoDB

Introduction to Indexes

- Indexes are essential for optimizing query performance in MongoDB.
- In MongoDB, indexes enhance the speed of query execution and can significantly improve overall database performance.

When Do We Need Indexes?

- Indexes are crucial when you're dealing with large collections and want to improve the efficiency of queries.

- **High Query Volume**: If your application frequently queries specific fields, indexes can significantly speed up these queries.

- **Sorting and Aggregation**: Indexes improve sorting and aggregation operations, common in reporting and analytics scenarios.

- **Join Operations**: Indexes can enhance join-like operations if you're working with related data in different collections.

Using the createIndex Method

- **Step 1: Connect to the MongoDB Server**
  Before creating an index, establish a connection to your MongoDB server using the appropriate driver.

- **Step 2: Choose a Collection.**
  Select the collection you want to create an index.

- **Step 3: Determine the Index Fields**
  Identify the fields that should be indexed. For example, we have a **“products”** collection with a name field.

- **Step 4: Use createIndex.**
  Use the **createIndex** method to create an index on the chosen field:

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

Note: The number 1 indicates ascending order. Use -1 for descending order.

Compound Indexes

Compound indexes involve multiple fields. They can significantly enhance query performance for complex queries that involve multiple fields. Let's create a compound index for our **“products”** collection:

- **Step 1: Determine Index Fields.**

Choose the fields that you frequently query or filter together. For instance, let's consider the category and price fields.

- **Step 2: Create the Compound Index**

```
db.products.createIndex({ category: 1, price: -1 });
```

Use Cases for Compound Indexes

Imagine you're building an e-commerce platform. Here are a couple of scenarios where the compound index we created could be beneficial:

- **Filtering by Category and Price Range:** If users often search for products within a specific category and price range, the compound index will speed up these queries.

- **Sorting by Price Within a Category:** When users want to see products in a particular category sorted by price, the compound index will optimize this sorting operation.

Understanding Comparison and Logical Operators in MongoDB

Comparison and logical operators are essential tools for querying data in MongoDB. They allow you to filter, combine, and manipulate data to retrieve the exact information you need.

Comparison Operators

Comparison operators help you compare values and retrieve documents that match specific conditions. Here are some common comparison operators:

- **$eq**: Equals
- **$ne**: Not Equals

- **$gt**: Greater Than
- **$lt**: Less Than
- **$gte**: Greater Than or Equal
- **$lte**: Less Than or Equal

Use Case:

Imagine you have an e-commerce database with a ‘products’ collection. You want to find products with prices between $50 and $100.

```
db.products.find({ price: { $gte: 50, $lte: 100 } });
```

Logical Operators

Logical operators allow you to combine multiple conditions in your queries. Here are some common logical operators:

- **$and**: Logical AND
- **$or**: Logical OR
- **$not**: Logical NOT

Use Case:

Finding Premium or Discounted Products

Suppose you want to find products that are either premium products (price > $200) or products on discount (price < $50).

```
db.products.find({
  $or: [
    { price: { $gt: 200 } }, // Premium products
    { price: { $lt: 50 } }   // Discounted products
  ]
});
```

Combining Comparison and Logical Operators

You can combine comparison and logical operators to create more complex queries.

Use Case:
Premium Products with High Ratings

To find premium products (price > $200) with a high rating (rating > 4):

```
db.products.find({
  $and: [
    { price: { $gt: 200 } }, // Premium products
    { rating: { $gt: 4 } }   // High rating
  ]
});
```

Projection Operators in MongoDB

Projection operators are powerful tools in MongoDB that allow you to control which fields to include or exclude in query results. They provide flexibility in shaping query outputs to match your specific needs. Let's delve into how projection operators work and explore real-world use cases.

Basic Projection

The basic projection involves specifying which fields you want to retrieve from the documents in your query results. Here are the key projection operators:

- **{ field: 1 }**: Include the specified field.
- **{ _id: 0 }**: Exclude the _id field.

Use Case: Retrieving Specific Fields

Imagine you have a ‘users’ collection with various fields, but you only need the username and email of each user.

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

Nested Fields Projection

Projection operators work with nested fields as well, allowing you to extract specific subfields from documents.

Use Case: Extracting Address Information

Consider a ‘customers’ collection with nested address subdocuments. You're interested in only the city and state fields.

```
db.customers.find({}, { "address.city": 1, "address.state": 1,
_id: 0 });
```

Conditional Projection

Projection operators can be combined with query conditions to project fields conditionally.

Use Case: Showing Premium Users' Email Addresses

Suppose you have a subscribers collection and want to display email addresses only for users with premium subscriptions.

```
db.subscribers.find({ isPremium: true }, { email: 1, _id: 0 })
```

Aggregation Operators in MongoDB

Aggregation operators are a versatile toolset in MongoDB that allows you to process and transform data to gain insights and perform complex operations. They enable you to manipulate, reshape, and summarize data within your collections. Let's explore how aggregation operators work and delve into practical scenarios.

Basic Aggregation

The basic aggregation operation involves stages that process documents in sequence. Here's an overview of some key aggregation stages:

- **$match**: Filters documents based on specified criteria.
- **$group**: Groups documents by specific fields and performs aggregate calculations.
- **$project**: Shapes the output documents by including or excluding fields.
- **$sort**: Sorts documents based on specified fields.

Use Case: Calculate Average Rating

Consider a ‘products’ collection with name, category, and rating fields. You want to calculate the average rating for each category.

```
db.products.aggregate([
  {
    $group: {
      _id: "$category",
      avgRating: { $avg: "$rating" }
    }
  }
]);
```

Combining Aggregation Stages

You can chain multiple aggregation stages to perform more complex operations.

Use Case: Find Top Categories by Average Price

Given the same products collection, you want to find the top categories with the highest average price.

```
db.products.aggregate([
  {
    $group: {
      _id: "$category",
      avgPrice: { $avg: "$price" }
    }
  },
  {
    $sort: { avgPrice: -1 }
  },
  {
    $limit: 5
  }
]);
```

Aggregation Expressions

Aggregation expressions enable advanced calculations and transformations.

Use Case: Calculating Total Revenue

Assuming you have an ‘orders’ collection with quantity and price fields, you want to calculate the total revenue.

```
db.orders.aggregate([
  {
    $project: {
      totalRevenue: { $multiply: ["$quantity", "$price"] }
    }
  },
  {
    $group: {
      _id: null,
      total: { $sum: "$totalRevenue" }
    }
  }
]);
```

Transaction Operators in MongoDB

Transaction operators are essential tools in MongoDB for ensuring data consistency and integrity in multi-step operations. Transactions enable you to group multiple operations into a single unit of work that either completes entirely or leaves no trace. Let's explore how to use transaction operators and understand their real-world applications.

Starting a Session

A session is a logical binding for a series of operations. To start a session, you use the startSession method.

Use Case: E-Commerce Order Processing

Imagine you're processing an order, which involves deducting the product quantity and updating the order status. A session ensures these operations succeed together.

```
const session = client.startSession();
```

Starting a Transaction

Transactions are used to group multiple operations as a single atomic unit. You start a transaction using the startTransaction method within a session.

Use Case: Money Transfer

Suppose you're transferring money between accounts. You want to deduct from one account and credit to another, ensuring that both actions are completed or none at all.

```
session.startTransaction();
```

Committing a Transaction

To make the changes within a transaction permanent, you commit the transaction using the commitTransaction method.

Use Case: Reservation System

In a reservation system, you're booking seats for a concert. The reservation is only confirmed when payment is successful and the transaction is committed.

```
session.commitTransaction();
```

Aborting a Transaction

If a transaction encounters an issue, you can abort it to discard any changes using the abortTransaction method.

Use Case: Online Store Checkout

During checkout, if a user's payment fails, you'd want to abort the transaction to prevent changes to the order and inventory.

```
session.abortTransaction();
```

Ending a Session

Once you've completed all necessary operations, you can end the session using the endSession method.

Use Case: User Registration

After a user registers, you might have multiple operations like sending emails, creating profiles, and more. An ended session ensures these operations conclude.

```
session.endSession();
```

Closing the Client

To ensure proper resource management, close the client when you're done with all operations.

Use Case: Application Shutdown

When your application is shutting down or no longer needs the MongoDB connection, closing the client ensures graceful termination.

```
client.close();
```

Example of combining different operations within a transaction:

```
// Import the necessary MongoDB driver
const { MongoClient } = require("mongodb");

// Connection URL
const uri = "mongodb://localhost:27017";

// Create a new MongoClient
const client = new MongoClient(uri);

// Define the main function
async function main() {
  try {
    // Connect to the MongoDB server
    await client.connect();
    console.log("Connected to MongoDB");

    // Start a session
    const session = client.startSession();

    // Define the database and collection
    const database = client.db("mydb");
    const collection = database.collection("transactions");

    // Start a transaction
    session.startTransaction();

    try {
      // Insert a document
      await collection.insertOne({ name: "Transaction 1" });
      console.log("Document inserted");

      // Update the document
      await collection.updateOne({ name: "Transaction 1" }, {
$set: { status: "completed" } });
      console.log("Document updated");

      // Commit the transaction
      await session.commitTransaction();
      console.log("Transaction committed");
    } catch (error) {
      // If there's an error, abort the transaction
      console.log("Error:", error);
      console.log("Transaction aborted");
      await session.abortTransaction();
    } finally {
      // End the session
      session.endSession();
    }
  } catch (error) {
    console.log("Error:", error);
  } finally {
    // Close the client
    await client.close();
    console.log("MongoDB connection closed");
  }
}
```

Explanation:

- We import the necessary MongoClient from the MongoDB driver.
- Define the connection URL **(URI)** to your MongoDB server.
- Create a new instance of MongoClient.
- Define the main function where all the MongoDB operations take place.
- Inside the main function, we start by connecting to the MongoDB server using await **client.connect().**
- We start a session using **const session = client.startSession()**.
- Define the database and collection you want to work with using **client.db("mydb")** and **database.collection("transactions")**.
- Begin a transaction with **session.startTransaction()**.
- Inside the transaction, we perform two operations: inserting a document and updating its status in the collection.
- If the operations within the transaction are successful, we commit the transaction using **session.commitTransaction()**.

- If there's an error during the transaction, we handle it by printing the error, aborting the transaction with **session.abortTransaction()**, and then finally ending the session.

- After all the transaction handling, we close the MongoDB client connection using await **client.close()**.