1. Start the mongod server
2. Open another command priompt type mongosh

Collection-table

Document- row

**db.createCollection("products")**

{ ok: 1 }

test>

**db.createCollection("products")**

**db.products.insertOne({"name":"Boost","price":"320","category":"edible"})**

{

acknowledged: true,

insertedId: **ObjectId('66bc2e3e743383ce3478c1e7')**

}

test>

**db.createCollection("products")**

**db.products.insertOne({"name":"Boost","price":"320","category":"edible"})**

**db.products.find();**

[

{

\_id: ObjectId('66bc2e3e743383ce3478c1e7'),

name: 'Boost',

price: '320',

category: 'edible'

}

]

**db.createCollection("products")**

**db.products.insertOne({"name":"Boost","price":"320","category":"edible"})**

**db.products.find();**

**db.products.insertMany([{"name":"TV","price":"32000","category":"electronics"},{"name":"watch"}])**

{

acknowledged: true,

insertedIds: {

'0': ObjectId('66bc2f65743383ce3478c1e8'),

'1': ObjectId('66bc2f65743383ce3478c1e9')

}

}

test> **db.products.find()**

[

{

\_id: ObjectId('66bc2e3e743383ce3478c1e7'),

name: 'Boost',

price: '320',

category: 'edible'

},

{

\_id: ObjectId('66bc2f65743383ce3478c1e8'),

name: 'TV',

price: '32000',

category: 'electronics'

},

{ \_id: ObjectId('66bc2f65743383ce3478c1e9'), name: 'watch' }

]

**db.createCollection("products")**

**db.products.insertOne({"name":"Boost","price":"320","category":"edible"})**

**db.products.find();**

**db.products.insertMany([{"name":"TV","price":"32000","category":"electronics"},{"name":"watch"}])**

**db.products.find({},{name:1})**

[

{ \_id: ObjectId('66bc2e3e743383ce3478c1e7'), name: 'Boost' },

{ \_id: ObjectId('66bc2f65743383ce3478c1e8'), name: 'TV' },

{ \_id: ObjectId('66bc2f65743383ce3478c1e9'), name: 'watch' }

]

**db.createCollection("products")**

**db.products.insertOne({"name":"Boost","price":"320","category":"edible"})**

**db.products.find();**

**db.products.insertMany([{"name":"TV","price":"32000","category":"electronics"},{"name":"watch"}])**

**db.products.find({},{name:1})**

test> **db.products.find({name:{$eq:"TV"}})**

[

{

\_id: ObjectId('66bc2f65743383ce3478c1e8'),

name: 'TV',

price: '32000',

category: 'electronics'

}

]

db.createCollection(posts)

db.posts.insertMany([

{

title: "Post Title 2",

body: "Body of post.",

category: "Event",

likes: 2,

tags: ["news", "events"],

date: Date()

},

{

title: "Post Title 3",

body: "Body of post.",

category: "Technology",

likes: 3,

tags: ["news", "events"],

date: Date()

},

{

title: "Post Title 4",

body: "Body of post.",

category: "Event",

likes: 4,

tags: ["news", "events"],

date: Date()

}

])

Find()

‘db.products.find()

‘db.products.findone() will return fist document found

‘querying data

Db.posts.find({})

Using where

db.posts.find({}, {title: 1, date: 1})

update

db.posts.updateOne(

{ title: "Post Title 1" },

{ $set: { likes: 2 }

} )

Update the document, but if not found insert it:

db.posts.updateOne(

{ title: "Post Title 5" },

{

$set:

{

title: "Post Title 5",

body: "Body of post.",

category: "Event",

likes: 2,

tags: ["news", "events"],

date: Date()

}

},

{ upsert: true }

)

db.collection.find({ age: { $eq: 30 } })

db.collection.find({ age: { $gt: 30 } })

db.collection.find({ $and: [

{ age: { $gt: 20 } },

{ age: { $lt: 40 } } ]

})

db.collection.find({ $or: [ { name: { $eq: “Horlicks”} }, { name { $eq: “Boost”} } ] })

db.collection.find({ email: { $exists: true } })// **$exists**: Matches documents that have the specified field.

db.collection.find({ name: { $regex: /^J/ } })

db.collection.find({ tags: { $all: ["mongodb", "nosql"] } })

**$elemMatch**: Matches documents that contain an array field with at least one element matching the specified query criteria.

db.collection.find({ scores: { $elemMatch: { $gte: 80, $lt: 90 } } })

[ { "\_id": ObjectId("66bc34d6743383ce3478c1ea"),

"title": "Post Title 2",

"tags": [ { "type": "news", "count": 5 }, { "type": "events", "count": 3 } ]

},

{ "\_id": ObjectId("66bc34d6743383ce3478c1eb"),

"title": "Post Title 3",

"tags": [ { "type": "news", "count": 2 }, { "type": "events", "count": 7 } ]

}

]

db.posts.find({

tags: {

$elemMatch: { count: { $gte: 5, $lt: 8 } }

}

})

CREATE COLLECTIONS

Step 1: use onlineshopping

steip 2: db.createCollection('product') ->it like an table

Step 3:db.product.insertMany([{name:"horlicks",qty:10,proce:250,category:"eatables"},{name:"abc",price:2500}]->insert data

Step 4:db.product.find()

step 5:db.product.findOne()->it show the first data

step 6:db.product.find({name:"abc"})

1.find the document where name = horilcks

ans:db.product.find({name:"horlicks"})

2.filter the drinkable the boost

ans:db.product.find({category:"drinkable",name:"boost"})

3.display the document where price is greater than 100

ans:db.product.find({price:{$gt:100}})

4.to filter only the products we have category

ans: db.product.find({category:{$exists:true}})

5.using regular expression in document where it contains abl

ans:db.product.find({category:{$regex:/abl/}})

6.Projection:{}->it like an select query

to display name and category

db.product.find({},{name:1,category:1})

7.$elemMatch-it check the array of data in a document.i specify the condition in one criteria

Comparison

The following operators can be used in queries to compare values:

* $eq: Values are equal
* $ne: Values are not equal
* $gt: Value is greater than another value
* $gte: Value is greater than or equal to another value
* $lt: Value is less than another value
* $lte: Value is less than or equal to another value
* $in: Value is matched within an array

Logical

The following operators can logically compare multiple queries.

* $and: Returns documents where both queries match
* $or: Returns documents where either query matches
* $nor: Returns documents where both queries fail to match
* $not: Returns documents where the query does not match

Evaluation

The following operators assist in evaluating documents.

* $regex: Allows the use of regular expressions when evaluating field values
* $text: Performs a text search
* $where: Uses a JavaScript expression to match documents

**1.$in example**

[{

"\_id" : 1,

"name" : "Product 1",

"tags" : ["electronics", "smartphone"]

},

{

"\_id" : 2,

"name" : "Product 2",

"tags" : ["electronics", "laptop"]

},

{

"\_id" : 3,

"name" : "Product 3",

"tags" : ["clothing", "jacket"]

}]

**db.products.find({ "tags": { $in: ["electronics", "laptop"] } })**

const mongoose = require('mongoose');

// Define schema

const productSchema = new mongoose.Schema({

name: String,

tags: [String]

});

// Define model

const Product = mongoose.model('Product', productSchema);

// Connect to MongoDB

mongoose.connect('mongodb://localhost:27017/mydatabase', { useNewUrlParser: true, useUnifiedTopology: true })

.then(() => {

console.log("Connected to MongoDB");

// Find products with tags "electronics" or "laptop"

Product.find({ tags: { $in: ["electronics", "laptop"] } })

.then(products => {

console.log("Products with tags 'electronics' or 'laptop':");

console.log(products);

})

.catch(err => console.error(err))

.finally(() => mongoose.disconnect());

})

.catch(err => console.error("Error connecting to MongoDB:", err));

**2.$text**

db.sample.insertMany[{

"\_id" : 1,

"title" : "The Great Gatsby",

"author" : "F. Scott Fitzgerald",

"description" : "The story primarily concerns the young and mysterious millionaire Jay Gatsby and his quixotic passion and obsession for the beautiful former debutante Daisy Buchanan."

},

{

"\_id" : 2,

"title" : "To Kill a Mockingbird",

"author" : "Harper Lee",

"description" : "The novel is renowned for its warmth and humor, despite dealing with the serious issues of rape and racial inequality."

},

{

"\_id" : 3,

"title" : "1984",

"author" : "George Orwell",

"description" : "It follows the life of Winston Smith, a low-ranking member of the ruling Party, in a dystopian London."

}]

db.sample.createIndex({name:”text”,title:”text”})

db.sample.getIndexes()

db.books.find({ $text: { $search: "novel" } })

[{

"\_id" : 1,

"order\_number" : "ORD-001",

"order\_date" : ISODate("2024-02-27T00:00:00Z")

},

{

"\_id" : 2,

"order\_number" : "ORD-002",

"order\_date" : ISODate("2024-02-26T00:00:00Z")

}]

**db.orders.updateOne(**

**{ "order\_number": "ORD-001" },**

**{ $currentDate: { "order\_date": true } }**

**)**

Update

db.inventory.updateOne(

{ \_id: 1 }, // Query criteria, in this case, we are updating the document with \_id equal to 1

{ $inc: { quantity: 10 } } // Increment the quantity field by 10

);

db.inventory.updateMany(

{}, // Empty query to match all documents

{ $rename: { "description": "product\_description" } } // Rename the 'description' field to 'product\_description'

);

db.inventory.updateOne(

{ \_id: 2 }, // Query criteria, in this case, we are updating the document with \_id equal to 2

{ $set: { price: 29.99 } } // Set the price field to 29.99

);

db.inventory.updateOne(

{ \_id: 3 }, // Query criteria, in this case, we are updating the document with \_id equal to 3

{ $unset: { product\_description: "" } } // Remove the 'product\_description' field

);

**Array**

Suppose you have a collection called **users** with documents representing users, and each document has an array field **interests** containing the user's interests. You want to add a new interest to the interests array, ensuring it's unique:

**db.users.updateOne(**

**{ \_id: 1 }, // Query criteria, in this case, we are updating the document with \_id equal to 1**

**{ $addToSet: { interests: "cooking" } } // Add "cooking" to the interests array if it doesn't already exist**

**);**

**db.users.updateOne(**

**{ \_id: 1 }, // Query criteria, in this case, we are updating the document with \_id equal to 1**

**{ $pop: { interests: 1 } } // Remove the last element from the interests array**

**);**

**db.users.updateOne(**

**{ \_id: 1 }, // Query criteria, in this case, we are updating the document with \_id equal to 1**

**{ $pull: { interests: "reading" } } // Remove all occurrences of "reading" from the interests array**

**);**

**db.users.updateOne(**

**{ \_id: 1 }, // Query criteria, in this case, we are updating the document with \_id equal to 1**

**{ $push: { interests: "gardening" } } // Add "gardening" to the interests array**

**);**

**\***

**Aggregation**

**$out**

The first stage will group properties by the property\_type and include the name, accommodates, and price fields for each. The $out stage will create a new collection called properties\_by\_type in the current database and write the resulting documents into that collection.

**MongoDB Model**

In MongoDB, a data model refers to the structure and organization of the data within a database. MongoDB is a NoSQL database, which means it does not require a predefined schema like traditional SQL databases. Instead, MongoDB allows for flexible schema design, enabling you to store heterogeneous data with varying structures within the same collection.

The key components of a data model in MongoDB include:

1. **Collections**: Collections are analogous to tables in relational databases. They contain individual documents, which are equivalent to rows in a table. Collections in MongoDB can store documents with varying structures.
2. **Documents**: Documents are JSON-like data structures stored in BSON (Binary JSON) format. They represent a single record within a collection. Documents can have nested fields and arrays, allowing for rich and flexible data structures.
3. **Fields**: Fields are key-value pairs within a document. Each field has a name and a value. Values can be of various types, including strings, numbers, arrays, nested documents, and more.
4. **Indexes**: Indexes are optional but highly recommended for efficient querying. They allow MongoDB to quickly locate documents based on the values of one or more fields. MongoDB supports various types of indexes, including single-field, compound, multikey, and geospatial indexes.

{

"\_id": ObjectId("61736fbfc5f7fe188ec5c0b1"),

"title": "To Kill a Mockingbird",

"author": "Harper Lee",

"genre": ["Fiction", "Classic"],

"publication\_year": 1960,

"isbn": "9780061120084"

}

// Find books by title

db.books.find({ "title": "To Kill a Mockingbird" });

// Find books by author

db.books.find({ "author": "Harper Lee" });

1. **db.books.createIndex({ "title": 1 })** creates an ascending index on the "title" field.
2. **db.books.createIndex({ "author": 1 })** creates an ascending index on the "author" field.
3. **Traditional Indexes (e.g., on "title" and "author" fields)**:
   1. Traditional indexes are used for efficient querying based on the exact values of indexed fields.
   2. They are suitable for equality queries, range queries, and sorting.
   3. Traditional indexes are not specifically optimized for text search or querying based on textual content within string fields.
4. **$search Index**:
   1. **$search** indexes are designed specifically for performing full-text search operations.
   2. They enable you to search for text within string fields using various linguistic features such as stemming, stop words, and language-specific rules.
   3. **$search** indexes support text search operators like **$search**, **$text**, and **$language**.
   4. They are well-suited for searching text across one or more fields containing textual content, such as descriptions, articles, or comments.
   5. **$search** indexes provide scoring capabilities, allowing you to rank search results based on relevance.

In summary, while traditional indexes are suitable for efficient querying based on exact field values, **$search** indexes are specialized for performing full-text search operations, making them ideal for text search use cases where you need to search within textual content across one or more fields.

1. **Schema Design**: While MongoDB offers flexibility in schema design, it's still important to consider the access patterns and query requirements of your application. Proper schema design can optimize query performance and ensure data integrity.

Overall, the data model in MongoDB is designed to be flexible, scalable, and capable of handling diverse data structures and access patterns. By leveraging collections, documents, fields, indexes, and thoughtful schema design, you can effec

**MngoDB Datatype**

MongoDB supports various data types to accommodate different types of data in documents. Here are some of the commonly used data types in MongoDB:

1. **String**: Represents UTF-8 encoded strings. Strings are commonly used for storing textual data.
2. **Integer**: Represents 32-bit signed integers.
3. **Double**: Represents 64-bit floating-point numbers.
4. **Boolean**: Represents boolean values **true** and **false**.
5. **Date**: Represents a date and time. Dates are stored as BSON datetime objects.
6. **ObjectId**: Represents a unique identifier for documents. ObjectId values are generated by MongoDB to uniquely identify each document in a collection.
7. **Array**: Represents an ordered list of values. Arrays can contain values of different data types, including other arrays and documents.
8. **Embedded Document**: Represents nested documents within a document. Embedded documents allow for hierarchical data structures.
9. **Binary Data**: Represents binary data, such as files or images. Binary data is stored as BSON binary objects.
10. **Null**: Represents a null value.
11. **Regular Expression**: Represents a regular expression pattern.
12. **Timestamp**: Represents a timestamp for operations within a replica set. Timestamp values are typically generated by MongoDB drivers for internal use.
13. **Decimal128**: Represents a 128-bit decimal floating-point number. Decimal128 is suitable for representing decimal values with high precision.
14. **MinKey and MaxKey**: Represents the lowest and highest possible BSON key values, respectively. These are typically used as placeholders or sentinels.

BSON (Binary JSON) is a way to serialize and store JSON-like documents in binary format, which MongoDB uses to store its data efficiently. In MongoDB, each document is represented in BSON format.

Here's a breakdown of BSON types with examples:

**{ "price": 9.99 }**

**{ "name": "John Doe" }**

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

**{ "tags": ["mongodb", "database", "nosql"] }**

In MongoDB, every document stored in a collection must have a unique identifier. This identifier is stored in a field called **\_id**. The **\_id** field serves as the primary key for the document and ensures its uniqueness within the collection.

Here are some key points about the **\_id** field in MongoDB:

1. **Uniqueness**: The **\_id** field must be unique within the collection. No two documents in the same collection can have the same **\_id** value.
2. **Automatic Generation**: If you don't specify an **\_id** field when inserting a document, MongoDB automatically generates a unique identifier for it. By default, MongoDB uses a BSON ObjectID as the value for the **\_id** field, which is a 12-byte hexadecimal value consisting of a timestamp, machine identifier, process identifier, and a random incrementing value.
3. **Custom \_id Values**: You can also specify your own unique values for the **\_id** field. These values can be of any BSON data type, but they must be unique within the collection.
4. **Indexing**: MongoDB automatically creates an index on the **\_id** field to facilitate fast retrieval of documents by their unique identifier. This ensures efficient queries based on the **\_id** field.

Here's an example of a document in MongoDB with an **\_id** field

**{**

**"\_id": ObjectId("6170822b97e2a6b97b013a9f"),**

**"name": "John Doe",**

**"age": 30,**

**"email": "john.doe@example.com"**

**}**

In MongoDB, a document is a data structure composed of field-and-value pairs. It is the basic unit of data storage, similar to a row in a relational database table. Documents are stored in collections, which are analogous to tables in relational databases.

Here are some key characteristics of documents in MongoDB:

1. **Flexible Schema**: MongoDB is a NoSQL database, which means it does not require a predefined schema. Documents within a collection can have different structures and fields. This flexibility allows for easy adaptation to changing data requirements.
2. **JSON-like Format**: Documents in MongoDB are typically represented in a JSON-like format called BSON (Binary JSON), which stands for Binary JSON. BSON extends JSON to include additional data types and to support efficient binary encoding and decoding.
3. **Field-and-Value Pairs**: Each document consists of one or more field-and-value pairs, where a field is a unique identifier for a value within the document. The value can be of various BSON data types, including strings, numbers, arrays, nested documents, binary data, and more.
4. **\_id Field**: Every document in a collection must contain a special field called **\_id**, which serves as the primary key for the document. The **\_id** field uniquely identifies each document within the collection.
5. **Atomicity**: Operations on a single document in MongoDB are atomic, meaning they either fully complete or have no effect at all. This atomicity is guaranteed for operations such as inserts, updates, and deletes.

Here's an example of a document in MongoDB:

**{**

**"\_id": ObjectId("6170822b97e2a6b97b013a9f"),**

**"name": "John Doe",**

**"age": 30,**

**"email": "john.doe@example.com",**

**"address": {**

**"city": "New York",**

**"zip": "10001"**

**},**

**"tags": ["mongodb", "database", "nosql"]**

**}**

**textIndex Vs SearchIndex**

If you create an index on the "username" field using **{ "username": 1 }**, it will be a standard ascending index rather than a text index. While this index can still be used to search for documents based on the "username" field, it won't support full-text search capabilities like the text index does.

You can still use this index for searching, but you would need to use equality or range queries rather than text search queries. For example:

**// Assuming you've created the index**

**db.users.createIndex({ "username": 1 });**

**// Searching for a specific username**

**const usernameToSearch = "john\_doe";**

**const searchResult = db.users.find({ "username": usernameToSearch });**

**// Iterate over the search result**

**searchResult.forEach(user => {**

**print(`User found: ${user.name} (${user.email})`);**

**});**

In this case, the index on the "username" field will help MongoDB efficiently locate documents with the specified username. However, this approach is more suitable for exact matches and may not be as effective for text search operations or partial matches compared to using a text index.

Top of Form

when creating a text index, the value **"text"** is always specified as it indicates that the index should be of type text. Unlike other index types where you might specify options or configurations, the text index is specifically designated as **"text"** to differentiate it from other types of indexes.

So, in the command:

**db.users.createIndex({ "username": "text" });**

The value **"text"** is not referring to the content of the "username" field, but rather to the type of index being created. It instructs MongoDB to create a text index on the "username" field, enabling full-text search capabilities on that field.

Once you've created a unique index on the "username" field with the query **db.collection.createIndex({ "username": 1 }, { unique: true })**, you can perform searches based on the "username" field efficiently. Here's an example of how to search for a document using the "username" field:

// Assume we have a collection named "users"

// First, let's insert some sample documents

db.users.insertMany([

{ "username": "john\_doe", "name": "John Doe", "email": "john@example.com" },

{ "username": "jane\_smith", "name": "Jane Smith", "email": "jane@example.com" },

{ "username": "alice\_wonder", "name": "Alice Wonderland", "email": "alice@example.com" }

]);

// Now, let's search for a user by their username

const usernameToSearch = "john\_doe";

const user = db.users.findOne({ "username": usernameToSearch });

if (user) {

print(`User found: ${user.name} (${user.email})`);

} else {

print(`User with username "${usernameToSearch}" not found.`);

}

In this example, we're using the **findOne()** method to search for a user document with the specified username ("john\_doe"). Since we've created a unique index on the "username" field, MongoDB can quickly locate the document corresponding to the given username.

If a document with the specified username is found, its details (name and email) are printed. Otherwise, a message indicating that the user with the specified username was not found is printed.

Unique indexes are particularly useful for enforcing uniqueness constraints on fields and ensuring efficient retrieval of documents based on their unique values, such as usernames in this example.

In MongoDB, the **$search** operator is used with the aggregation framework's **$match** stage to perform text search queries. However, it's important to note that the **$search** operator is available in MongoDB Atlas Full-Text Search, which requires a MongoDB Atlas cluster and is not available in all MongoDB installations.

Assuming you're using MongoDB Atlas Full-Text Search and you want to search for users based on their usernames, here's how you can apply the **$search** operator:

**// Assume we have a collection named "users" with a text index on the "username" field**

**// First, let's insert some sample documents**

**db.users.insertMany([**

**{ "username": "john\_doe", "name": "John Doe", "email": "john@example.com" },**

**{ "username": "jane\_smith", "name": "Jane Smith", "email": "jane@example.com" },**

**{ "username": "alice\_wonder", "name": "Alice Wonderland", "email": "alice@example.com" }**

**]);**

**// Now, let's search for users by their usernames using the $search operator**

**const usernameToSearch = "john\_doe";**

**const searchResult = db.users.aggregate([**

**{**

**$search: {**

**"text": {**

**"query": usernameToSearch,**

**"path": "username"**

**}**

**}**

**}**

**]);**

**searchResult.forEach(user => {**

**print(`User found: ${user.name} (${user.email})`);**

**});**

Indexes in databases, including MongoDB, are data structures that help optimize query performance by allowing for efficient data retrieval. Understanding the properties of indexes is crucial for designing and using them effectively. Here are some key properties of indexes:

1. **Search Efficiency**: Indexes facilitate quick data retrieval by allowing the database to locate relevant documents or rows without scanning the entire collection or table. This improves query performance, especially for large datasets.
2. **Data Organization**: Indexes organize data in a specific order, such as ascending or descending, based on the indexed field(s). This organization helps speed up search operations and enables efficient range queries and sorting.
3. **Query Optimization**: Indexes optimize query execution by providing the database with information on how to access and traverse data efficiently. They enable the query planner to choose the most appropriate index for a given query, leading to faster query execution times.
4. **Unique Constraint**: Some indexes enforce uniqueness constraints on fields, ensuring that no two documents or rows in a collection or table can have the same value for the indexed field(s). Unique indexes help maintain data integrity and prevent duplicates.
5. **Data Modification Overhead**: While indexes improve read performance, they can introduce overhead during data modification operations such as inserts, updates, and deletes. This overhead arises from the need to maintain index structures and keep them in sync with the underlying data.
6. **Index Size**: Indexes consume additional storage space to store index keys and metadata. The size of an index depends on factors such as the number of indexed fields, the cardinality of the indexed values, and the index type.
7. **Index Types**: Different types of indexes serve various purposes and have specific characteristics. Common index types include single-field indexes, compound indexes, text indexes, geo-spatial indexes, hashed indexes, and others. Each index type has its own advantages and limitations, and choosing the right index type depends on the nature of the data and the types of queries performed.
8. **Index Maintenance**: Indexes require periodic maintenance to ensure optimal performance. This includes tasks such as index rebuilding, index optimization, and monitoring index usage patterns. Regular maintenance helps prevent index fragmentation and degradation of query performance over time.
9. **Index Selectivity**: Index selectivity refers to the uniqueness or distinctiveness of the indexed values within a collection or table. Highly selective indexes have a wide range of unique values, making them more efficient for filtering and narrowing down search results.
10. **Query Coverage**: Indexes can cover query predicates, sort orders, and projection fields, allowing queries to be fully satisfied by accessing the index without needing to access the underlying documents or rows. This enhances query performance by reducing the need for disk I/O operations.

Understanding these properties of indexes is essential for effective database schema design, query optimization, and performance tuning in MongoDB and other database systems. Properly leveraging indexes can significantly improve the efficiency and responsiveness of database operations.

**Understanding the properties of indexes**

Indexes in databases, including MongoDB, are data structures that help optimize query performance by allowing for efficient data retrieval. Understanding the properties of indexes is crucial for designing and using them effectively. Here are some key properties of indexes:

1. **Search Efficiency**: Indexes facilitate quick data retrieval by allowing the database to locate relevant documents or rows without scanning the entire collection or table. This improves query performance, especially for large datasets.
2. **Data Organization**: Indexes organize data in a specific order, such as ascending or descending, based on the indexed field(s). This organization helps speed up search operations and enables efficient range queries and sorting.
3. **Query Optimization**: Indexes optimize query execution by providing the database with information on how to access and traverse data efficiently. They enable the query planner to choose the most appropriate index for a given query, leading to faster query execution times.
4. **Unique Constraint**: Some indexes enforce uniqueness constraints on fields, ensuring that no two documents or rows in a collection or table can have the same value for the indexed field(s). Unique indexes help maintain data integrity and prevent duplicates.
5. **Data Modification Overhead**: While indexes improve read performance, they can introduce overhead during data modification operations such as inserts, updates, and deletes. This overhead arises from the need to maintain index structures and keep them in sync with the underlying data.
6. **Index Size**: Indexes consume additional storage space to store index keys and metadata. The size of an index depends on factors such as the number of indexed fields, the cardinality of the indexed values, and the index type.
7. **Index Types**: Different types of indexes serve various purposes and have specific characteristics. Common index types include single-field indexes, compound indexes, text indexes, geo-spatial indexes, hashed indexes, and others. Each index type has its own advantages and limitations, and choosing the right index type depends on the nature of the data and the types of queries performed.
8. **Index Maintenance**: Indexes require periodic maintenance to ensure optimal performance. This includes tasks such as index rebuilding, index optimization, and monitoring index usage patterns. Regular maintenance helps prevent index fragmentation and degradation of query performance over time.
9. **Index Selectivity**: Index selectivity refers to the uniqueness or distinctiveness of the indexed values within a collection or table. Highly selective indexes have a wide range of unique values, making them more efficient for filtering and narrowing down search results.
10. **Query Coverage**: Indexes can cover query predicates, sort orders, and projection fields, allowing queries to be fully satisfied by accessing the index without needing to access the underlying documents or rows. This enhances query performance by reducing the need for disk I/O operations.

Understanding these properties of indexes is essential for effective database schema design, query optimization, and performance tuning in MongoDB and other database systems. Properly leveraging indexes can significantly improve the efficiency and responsiveness of database operations.

**Collection Name is StreamingData**

**[**

**{**

**"sensor\_id": "sensor001",**

**"timestamp": ISODate("2024-02-28T00:00:00Z"),**

**"value": 24.5,**

**"unit": "Celsius",**

**"location": {**

**"latitude": 40.7128,**

**"longitude": -74.0060**

**}**

**},**

**{**

**"sensor\_id": "sensor002",**

**"timestamp": ISODate("2024-02-28T00:01:00Z"),**

**"value": 982,**

**"unit": "Pascal",**

**"location": {**

**"latitude": 34.0522,**

**"longitude": -118.2437**

**}**

**},**

**{**

**"sensor\_id": "sensor003",**

**"timestamp": ISODate("2024-02-28T00:02:00Z"),**

**"value": 12.8,**

**"unit": "Meters per second",**

**"location": {**

**"latitude": 41.8781,**

**"longitude": -87.6298**

**}**

**},**

**{**

**"sensor\_id": "sensor001",**

**"timestamp": ISODate("2024-02-28T00:03:00Z"),**

**"value": 23.8,**

**"unit": "Celsius",**

**"location": {**

**"latitude": 40.7128,**

**"longitude": -74.0060**

**}**

**},**

**{**

**"sensor\_id": "sensor002",**

**"timestamp": ISODate("2024-02-28T00:04:00Z"),**

**"value": 979,**

**"unit": "Pascal",**

**"location": {**

**"latitude": 34.0522,**

**"longitude": -118.2437**

**}**

**},**

**{**

**"sensor\_id": "sensor003",**

**"timestamp": ISODate("2024-02-28T00:05:00Z"),**

**"value": 11.9,**

**"unit": "Meters per second",**

**"location": {**

**"latitude": 41.8781,**

**"longitude": -87.6298**

**}**

**}**

**]**

**1.Display sensorid and units**

**2.Display the data stremed before "2024-02-28T00:03:00Z"**

**3.Display the details whose latitude is 41.8781 and longitude is -876298 use location.latitude and location.longitude to refer data**

**4.display number of data stremed from each sensor**

**5.Display average value for each sensor sort by sensor\_id**

**6.Display sensor\_id whose unit is Celsius and Meters Per second**

**7.Display sensor id and whose maximum streamed value is greater than 100**

**Users Collection**

**[**

**{**

**"\_id": 1,**

**"username": "john\_doe",**

**"name": "John Doe",**

**"email": "john@example.com",**

**"birthdate": ISODate("1990-05-15"),**

**"followers": ["user002", "user003", "user004"],**

**"following": ["user005", "user006"],**

**"bio": "Software Engineer Passionate about technology",**

**"profile\_picture": "john\_doe.jpg"**

**},**

**{**

**"\_id": 2,**

**"username": "jane\_smith",**

**"name": "Jane Smith",**

**"email": "jane@example.com",**

**"birthdate": ISODate("1988-12-30"),**

**"followers": ["user001", "user003"],**

**"following": ["user001", "user004"],**

**"bio": "Travel enthusiast | Nature lover",**

**"profile\_picture": "jane\_smith.jpg"**

**},**

**{**

**"\_id": 3,**

**"username": "nature\_lover1",**

**"name": "Nature Lover 1",**

**"email": "nature1@example.com",**

**"birthdate": ISODate("1995-08-20"),**

**"followers": ["user004", "user005"],**

**"following": ["user001", "user002"],**

**"bio": "Exploring the beauty of nature | Adventure seeker",**

**"profile\_picture": "nature\_lover1.jpg"**

**},**

**{**

**"\_id": 4,**

**"username": "nature\_lover2",**

**"name": "Nature Lover 2",**

**"email": "nature2@example.com",**

**"birthdate": ISODate("1992-04-10"),**

**"followers": ["user001", "user003"],**

**"following": ["user002", "user005"],**

**"bio": "Nature photographer | Outdoor enthusiast",**

**"profile\_picture": "nature\_lover2.jpg"**

**}**

**]**

* 1. **Display Nature Lovers details**
  2. **User004 didn’t follow user002 update the details**
  3. **Update user2 email id with your company email id**
  4. **Display user003 followers details**
  5. **Display user details followed by userr003**
  6. **Display user003 followers birthdate**
  7. **Disply userid and their profile pic**
  8. **Display all photographer details**

**What is Replication?**

Replication is the process of synchronizing data across multiple servers. It provides redundancy and increases data availability eith multiple copies of data on different database server.

**Replicaset in MongoDB**

**Creating it in local machine**

**Introduction to Sharding**

Start mongod -replSet viji -logPath \data\rs3\3.log –dbpath \data\rs3 –port 27017

Start mongod -replSet viji -logPath \data\rs3\3.log –dbpath \data\rs3 –port 27018

Start mongod -replSet viji -logPath \data\rs3\3.log –dbpath \data\rs3 –port 27019

Mongo –port 27017

Mongo –port 27018

Mongo –port 27019

In 17

Rs.intitate()

In other 2

Rs.slave()

**Shardning**

Setting up sharding in MongoDB involves several steps. Here's a high-level overview of the process:

1. **Deploy a Sharded Cluster**:
   * Deploy a new MongoDB cluster with the desired number of shards. Each shard can be a standalone MongoDB instance or a replica set.
   * MongoDB Atlas provides an easy way to deploy and manage sharded clusters in the cloud. If you're using MongoDB Atlas, you can follow the steps provided in the Atlas dashboard to deploy a sharded cluster.
2. **Enable Sharding for a Database**:
   * Connect to the mongos instance (MongoDB router) using the MongoDB shell or a client tool.
   * Enable sharding for the database by running the **sh.enableSharding("<database>")** command, where **<database>** is the name of the database you want to shard.
3. **Choose a Shard Key**:
   * Choose a field or fields in your documents to use as the shard key. The shard key determines how data is distributed across shards.
   * Select a shard key that evenly distributes data and aligns with your application's access patterns.
   * It's important to choose a shard key carefully, as changing the shard key later can be challenging.
4. **Shard a Collection**:
   * Once sharding is enabled for the database, choose a collection to shard.
   * Use the **sh.shardCollection("<database>.<collection>", { "<shardKey>": 1 })** command to shard the collection, specifying the database, collection, and shard key.
   * This command distributes the collection's data across shards based on the shard key.
5. **Monitor and Manage Shards**:
   * Monitor the sharded cluster using MongoDB monitoring tools or third-party monitoring solutions.
   * Scale the cluster by adding or removing shards as needed to accommodate data growth and workload changes.
   * MongoDB provides commands and tools for managing sharded clusters, such as **sh.addShard()** and **sh.removeShard()**.
6. **Query Data**:
   * Once sharding is configured, clients can interact with the sharded cluster through the mongos instance.
   * Queries and operations are automatically routed to the appropriate shards based on the shard key.

It's important to plan and test your sharding configuration carefully to ensure optimal performance and scalability. Consider factors such as shard key selection, data distribution, hardware requirements, and monitoring and management strategies.

Additionally, MongoDB provides comprehensive documentation on sharding, including detailed guides and tutorials, which can help you navigate the process of setting up and managing a sharded cluster effectively.

Top of Form

**$lookup**

**db.orders.aggregate([**

**{**

**$lookup: {**

**from: "customers", // The collection to join with**

**localField: "customerId", // Field from the "orders" collection**

**foreignField: "\_id", // Field from the "customers" collection**

**as: "customer" // Name of the field to store the joined documents**

**}**

**}**

**])**

o organize your Express application better, it's a good practice to separate routes and schema definitions into different files. Here's how you can structure your project:

1. Create a Separate Schema File
2. Create a Separate Route File
3. Integrate Routes and Schema in the Main Application

Project Structure

myapp/

├── models/

│ └── student.js

├── routes/

│ └── students.js

├── app.js

└── package.json

1. Define the Schema in models/student.js

Create a models directory and define your schema in student.js.

// models/student.js

const mongoose = require("mongoose");

// Define the schema

const studentSchema = new mongoose.Schema({

name: String,

department: String,

});

// Define and export the model

const Student = mongoose.model("Student", studentSchema);

module.exports = Student;

2. Create Routes in routes/students.js

Create a routes directory and define your routes in students.js.

// routes/students.js

const express = require("express");

const router = express.Router();

const Student = require("../models/student");

// Route to fetch all students

router.get("/", async (req, res) => {

try {

const students = await Student.find();

res.json(students);

} catch (err) {

res.status(500).json({ message: err.message });

}

});

// Route to fetch a student by ID

router.get("/:id", async (req, res) => {

try {

const student = await Student.findById(req.params.id);

if (student) {

res.json(student);

} else {

res.status(404).json({ message: "Student not found" });

}

} catch (err) {

res.status(500).json({ message: err.message });

}

});

// Route to add a new student

router.post("/", async (req, res) => {

const student = new Student({

name: req.body.name,

department: req.body.department,

});

try {

const newStudent = await student.save();

res.status(201).json(newStudent);

} catch (err) {

res.status(400).json({ message: err.message });

}

});

// Route to update a student by ID

router.put("/:id", async (req, res) => {

try {

const updatedStudent = await Student.findByIdAndUpdate(

req.params.id,

{

name: req.body.name,

department: req.body.department,

},

{ new: true }

);

if (updatedStudent) {

res.json(updatedStudent);

} else {

res.status(404).json({ message: "Student not found" });

}

} catch (err) {

res.status(400).json({ message: err.message });

}

});

// Route to delete a student by ID

router.delete("/:id", async (req, res) => {

try {

const deletedStudent = await Student.findByIdAndDelete(req.params.id);

if (deletedStudent) {

res.json({ message: "Student deleted" });

} else {

res.status(404).json({ message: "Student not found" });

}

} catch (err) {

res.status(500).json({ message: err.message });

}

});

module.exports = router;

3. Set Up the Main Application in app.js

In your app.js, import and use the routes.

// app.js

const express = require("express");

const mongoose = require("mongoose");

const app = express();

const port = 3000;

const studentRoutes = require("./routes/students");

app.use(express.json());

// Connect to MongoDB

mongoose

.connect("mongodb://localhost:27017/test", {

useNewUrlParser: true,

useUnifiedTopology: true,

})

.then(() => console.log("Connected to MongoDB"))

.catch((err) => console.error("Error connecting to MongoDB:", err));

// Use the student routes

app.use("/students", studentRoutes);

app.listen(port, () => {

console.log(`Server is running at http://localhost:${port}`);

});

Key Points

* Separation of Concerns: By separating schema, routes, and application logic, your code becomes easier to maintain and understand.
* Modular Design: Makes it easier to manage different parts of your application and scale it as needed.
* Error Handling: Ensure proper error handling and status codes for better API responses.

This structure will help you keep your code organized and make it easier to extend and maintain your application.