**MongoDB Syntax**

Here's a list of common MongoDB commands along with their syntax:

**1. Show Databases**

show dbs

Lists all available databases.

**2. Switch to a Database**

use <database\_name>

Switches to the specified database. If it doesn't exist, it will be created.

**3. Show Collections**

show collections

Lists all collections in the current database.

**4. Create a Collection**

db.createCollection("<collection\_name>")

Creates a new collection in the current database.

**5. Insert a Document**

db.<collection\_name>.insertOne({<field1>: <value1>, <field2>: <value2>})

Inserts a single document into the specified collection.

**6. Insert Multiple Documents**

db.<collection\_name>.insertMany([{<field1>: <value1>, <field2>: <value2>}, {...}])

Inserts multiple documents into the specified collection.

**7. Find All Documents in a Collection**

db.<collection\_name>.find()

Retrieves all documents from the specified collection.

**8. Find Documents with a Query**

db.<collection\_name>.find({<field>: <value>})

Finds documents that match the query criteria.

**9. Update a Document**

db.<collection\_name>.updateOne(

{<field>: <value>}, // Filter

{$set: {<field\_to\_update>: <new\_value>}} // Update

)

Updates a single document that matches the filter criteria.

**10. Update Multiple Documents**

db.<collection\_name>.updateMany(

{<field>: <value>}, // Filter

{$set: {<field\_to\_update>: <new\_value>}} // Update

)

Updates multiple documents that match the filter criteria.

**11. Delete a Document**

db.<collection\_name>.deleteOne({<field>: <value>})

Deletes a single document that matches the filter criteria.

**12. Delete Multiple Documents**

db.<collection\_name>.deleteMany({<field>: <value>})

Deletes multiple documents that match the filter criteria.

**13. Count Documents in a Collection**

db.<collection\_name>.countDocuments()

Counts the number of documents in the specified collection.

**14. Create an Index**

db.<collection\_name>.createIndex({<field>: 1})

Creates an index on the specified field. Use 1 for ascending order or -1 for descending order.

**15. Drop a Collection**

db.<collection\_name>.drop()

Drops the specified collection from the database.

**16. Drop a Database**

db.dropDatabase()

Drops the current database.

**17. Aggregation**

db.<collection\_name>.aggregate([

{ $match: {<field>: <value>} },

{ $group: { \_id: "$<field>", total: { $sum: "$<another\_field>" } } }

])

Performs an aggregation operation on the specified collection.

**18. Find One Document**

db.<collection\_name>.findOne({<field>: <value>})

Finds a single document that matches the query criteria.

**19. Rename a Collection**

db.<collection\_name>.renameCollection("<new\_collection\_name>")

Renames a collection in the current database.

**20. List Indexes of a Collection**

db.<collection\_name>.getIndexes()

Lists all indexes of the specified collection.

Generate list of commands for MongoDB with syntax and real world examples problem like - customers and sales

Here’s a list of MongoDB commands with their syntax, along with real-world examples using collections named customers and sales:

**1. Show Databases**

show dbs

**Example**: Lists all available databases such as ecommerce, analytics, etc.

**2. Switch to a Database**

use ecommerce

**Example**: Switch to the ecommerce database, which might contain collections like customers, sales, etc.

**3. Show Collections**

show collections

**Example**: Lists all collections in the ecommerce database, such as customers, sales, products.

**4. Create a Collection**

db.createCollection("customers")

**Example**: Creates a customers collection to store customer information.

**5. Insert a Document**

db.customers.insertOne({

"customer\_id": 1,

"name": "John Doe",

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

"address": "123 Main St, Anytown, USA"

})

**Example**: Inserts a single customer record into the customers collection.

**6. Insert Multiple Documents**

db.sales.insertMany([

{

"sale\_id": 1001,

"customer\_id": 1,

"product": "Laptop",

"amount": 1200,

"date": "2024-08-01"

},

{

"sale\_id": 1002,

"customer\_id": 2,

"product": "Smartphone",

"amount": 800,

"date": "2024-08-02"

}

])

**Example**: Inserts multiple sales records into the sales collection.

**7. Find All Documents in a Collection**

db.customers.find()

**Example**: Retrieves all customer records from the customers collection.

**8. Find Documents with a Query**

db.sales.find({"customer\_id": 1})

**Example**: Finds all sales associated with customer\_id 1 in the sales collection.

**9. Update a Document**

db.customers.updateOne(

{"customer\_id": 1},

{$set: {"email": "john.newemail@example.com"}}

)

**Example**: Updates the email address of the customer with customer\_id 1.

**10. Update Multiple Documents**

db.sales.updateMany(

{"product": "Laptop"},

{$set: {"discount": 100}}

)

**Example**: Updates all sales records where the product is "Laptop" to include a discount of $100.

**11. Delete a Document**

db.customers.deleteOne({"customer\_id": 2})

**Example**: Deletes the customer record with customer\_id 2.

**12. Delete Multiple Documents**

db.sales.deleteMany({"amount": {$lt: 500}})

**Example**: Deletes all sales records where the amount is less than $500.

**13. Count Documents in a Collection**

db.customers.countDocuments()

**Example**: Counts the number of documents (customers) in the customers collection.

**14. Create an Index**

db.sales.createIndex({"date": 1})

**Example**: Creates an index on the date field in the sales collection to speed up queries based on date.

**15. Drop a Collection**

db.customers.drop()

**Example**: Drops the customers collection from the database.

**16. Drop a Database**

db.dropDatabase()

**Example**: Drops the current database, such as ecommerce.

**17. Aggregation**

db.sales.aggregate([

{ $match: {"product": "Laptop"} },

{ $group: { \_id: "$customer\_id", totalSpent: { $sum: "$amount" } } }

])

**Example**: Aggregates total spending on laptops by each customer.

**18. Find One Document**

db.customers.findOne({"customer\_id": 1})

**Example**: Finds the first customer document with customer\_id 1.

**19. Rename a Collection**

db.sales.renameCollection("transactions")

**Example**: Renames the sales collection to transactions.

**20. List Indexes of a Collection**

db.sales.getIndexes()

**Example**: Lists all indexes in the sales collection.

**How to create new database and new table in mongodb**

In MongoDB, databases and collections (equivalent to tables in relational databases) are created implicitly when you start using them. Here's how you can create a new database and a new collection:

**1. Create a New Database**

To create a new database in MongoDB, you simply switch to the new database using the use command. If the database doesn't exist, it will be created when you insert the first document into a collection.

**Syntax:**

use <database\_name>

**Example:**

use ecommerce

This command switches to the ecommerce database. If ecommerce doesn't exist, MongoDB creates it once a document is inserted into a collection.

**2. Create a New Collection (Table)**

After switching to the database, you can create a new collection by inserting a document into it. Collections are created implicitly with the first insertion.

**Syntax:**

db.<collection\_name>.insertOne({<field1>: <value1>, <field2>: <value2>})

**Example:**

db.customers.insertOne({

"customer\_id": 1,

"name": "Alice Smith",

"email": "alice.smith@example.com",

"address": "456 Elm St, Anytown, USA"

})

This command creates a customers collection in the ecommerce database (if it doesn’t already exist) and inserts the first document into it.

**3. Verify Creation**

* **To verify the database creation:**

show dbs

This will list all databases. Note that the new database will only appear if it contains at least one document in a collection.

* **To verify the collection creation:**

**show collections**

This will list all collections in the current database.

By following these steps, you create a new database and a new collection in MongoDB.

**How to list indexes?**

To list all indexes in a specific collection in MongoDB, you can use the getIndexes() method.

**Syntax**

db.<collection\_name>.getIndexes()

**Example**

Suppose you have a collection named customers and you want to list all indexes in this collection:

db.customers.getIndexes()

**Output**

This command will return a list of all indexes in the customers collection, including the default \_id index. The output might look something like this:

[

{

"v" : 2,

"key" : { "\_id" : 1 },

"name" : "\_id\_",

"ns" : "ecommerce.customers"

},

{

"v" : 2,

"key" : { "email" : 1 },

"name" : "email\_1",

"ns" : "ecommerce.customers"

}

]

**Explanation**

* **v**: The version of the index.
* **key**: The fields that are indexed, with 1 for ascending and -1 for descending.
* **name**: The name of the index.
* **ns**: The namespace, which is the combination of the database name and collection name.

How to Create index?

Creating an index in MongoDB is done using the createIndex() method on a collection. Indexes can be created on one or multiple fields, and you can specify whether the index should be in ascending or descending order.

**Syntax**

db.<collection\_name>.createIndex({ <field>: <order> })

* **<field>**: The field you want to index.
* **<order>**: Use 1 for ascending order or -1 for descending order.

**Examples**

**1. Create a Single-Field Index**

If you want to create an index on the email field in the customers collection in ascending order:

db.customers.createIndex({ "email": 1 })

This creates an index on the email field to optimize queries that filter or sort by email.

**2. Create a Compound Index**

A compound index is an index on multiple fields. If you want to create an index on both last\_name and first\_name in the customers collection:

db.customers.createIndex({ "last\_name": 1, "first\_name": 1 })

This creates an index on both last\_name and first\_name, which is useful for queries that filter or sort by these fields together.

**3. Create a Unique Index**

If you want to ensure that all values in the username field are unique:

db.customers.createIndex({ "username": 1 }, { unique: true })

This creates a unique index on the username field, preventing duplicate values in that field across the collection.

**4. Create a Text Index**

Text indexes are used for text search in string content. If you want to create a text index on the description field:

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

This allows for full-text search queries on the description field in the products collection.

**5. Create a Geospatial Index**

For indexing geospatial data, like locations stored in latitude and longitude fields:

db.locations.createIndex({ "location": "2dsphere" })

This is used for geospatial queries, such as finding documents within a certain distance of a point.

**Verify the Index**

To confirm that the index was created, use the following command:

db.<collection\_name>.getIndexes()

This command will list all indexes in the collection, including the newly created one.

**Why Create Indexes?**

Indexes improve the efficiency of search queries. Without indexes, MongoDB has to perform a collection scan, meaning it has to check every document in a collection to find the relevant ones. Indexes can greatly speed up this process, especially for large collections.

**Compound index syntax?**

A compound index in MongoDB is an index that includes multiple fields, allowing queries to be optimized for searches that involve multiple fields. You can create a compound index by specifying multiple field-order pairs in the createIndex() method.

**Syntax**

db.<collection\_name>.createIndex({ <field1>: <order1>, <field2>: <order2>, ... })

* **<field1>, <field2>**: The fields you want to include in the index.
* **<order1>, <order2>**: The order for each field, where 1 is ascending and -1 is descending.

**Example**

Suppose you have a customers collection, and you frequently run queries that involve both last\_name and first\_name. To create a compound index on these fields:

db.customers.createIndex({ "last\_name": 1, "first\_name": 1 })

This index is sorted first by last\_name in ascending order, and within each last\_name, by first\_name in ascending order.

**Compound Index with Mixed Sort Orders**

You can also create a compound index with mixed sort orders. For example, if you want to sort last\_name in ascending order and first\_name in descending order:

db.customers.createIndex({ "last\_name": 1, "first\_name": -1 })

**Using a Compound Index in Queries**

Once the compound index is created, MongoDB can use it to optimize queries that involve either both last\_name and first\_name or just last\_name. For example:

* **Query using both fields:**

db.customers.find({ "last\_name": "Doe", "first\_name": "John" })

* **Query using only the first indexed field:**

db.customers.find({ "last\_name": "Doe" })

**Limitations**

* MongoDB can use the index to optimize queries that include the fields in the same order as the index, starting from the leftmost field.
* If a query only uses first\_name and not last\_name, the index cannot be used efficiently unless last\_name is also included in the query.

Compound indexes are powerful for optimizing queries that involve multiple fields, especially when those queries are common in your application.

**How to apply aggregate operations on mongodb and how to import and export data into mongodb**

**1. Applying Aggregate Operations in MongoDB**

Aggregation in MongoDB allows you to process data records and return computed results. It is typically used to perform complex data analysis and transformation tasks. The aggregate() method is used to perform aggregation operations on a collection.

**Basic Syntax**

db.<collection\_name>.aggregate([

{ <stage1>: { <operator>: <expression> }},

{ <stage2>: { <operator>: <expression> }},

...

])

Each stage in the pipeline performs a specific operation on the data.

**Example: Aggregating Sales Data**

Suppose you have a sales collection with documents that contain fields like customer\_id, product, amount, and date. Here's how you can use the aggregate() method:

**a. Grouping by customer\_id and Summing amount**

db.sales.aggregate([

{ $group: { \_id: "$customer\_id", totalSpent: { $sum: "$amount" }}}

])

* **$group**: Groups documents by customer\_id.
* **$sum**: Sums up the amount for each customer\_id.

**b. Filtering with $match and then Grouping**

db.sales.aggregate([

{ $match: { product: "Laptop" }},

{ $group: { \_id: "$customer\_id", totalSpent: { $sum: "$amount" }}}

])

* **$match**: Filters documents to include only those where the product is "Laptop".
* **$group**: Groups the filtered documents by customer\_id and sums the amount.

**c. Sorting Results**

db.sales.aggregate([

{ $group: { \_id: "$customer\_id", totalSpent: { $sum: "$amount" }}},

{ $sort: { totalSpent: -1 }}

])

* **$sort**: Sorts the results by totalSpent in descending order.

**d. Limiting Results**

db.sales.aggregate([

{ $group: { \_id: "$customer\_id", totalSpent: { $sum: "$amount" }}},

{ $sort: { totalSpent: -1 }},

{ $limit: 5 }

])

* **$limit**: Limits the results to the top 5 customers by totalSpent.

**2. Importing Data into MongoDB**

You can use the mongoimport tool to import data into a MongoDB database from various formats like JSON, CSV, and TSV.

**Basic Syntax**

mongoimport --db <database\_name> --collection <collection\_name> --file <path\_to\_file> --type <file\_type> --headerline

**Example: Importing a JSON File**

mongoimport --db ecommerce --collection customers --file /path/to/customers.json --jsonArray

* **--db**: Specifies the database name.
* **--collection**: Specifies the collection name.
* **--file**: Path to the file containing the data.
* **--jsonArray**: Indicates that the file is a JSON array.

**Example: Importing a CSV File**

mongoimport --db ecommerce --collection sales --type csv --headerline --file /path/to/sales.csv

* **--type csv**: Specifies that the file is a CSV.
* **--headerline**: Uses the first line of the CSV file as field names.

**3. Exporting Data from MongoDB**

You can use the mongoexport tool to export data from a MongoDB collection into various formats like JSON and CSV.

**Basic Syntax**

mongoexport --db <database\_name> --collection <collection\_name> --out <output\_file> --type <file\_type>

**Example: Exporting to a JSON File**

mongoexport --db ecommerce --collection customers --out /path/to/customers.json --jsonArray

* **--out**: Specifies the output file path.
* **--jsonArray**: Exports data as a JSON array.

**Example: Exporting to a CSV File**

mongoexport --db ecommerce --collection sales --type=csv --out /path/to/sales.csv --fields "sale\_id,customer\_id,product,amount,date"

* **--fields**: Specifies the fields to be exported, in the order they should appear in the CSV file.

**Summary**

* **Aggregation Operations**: Use the aggregate() method to perform operations like $match, $group, $sort, $limit, etc.
* **Importing Data**: Use mongoimport with options like --db, --collection, --file, and --type.
* **Exporting Data**: Use mongoexport with similar options to export data to JSON, CSV, or other formats.