**DDL (Data Definition Language) Commands in MongoDB**

DDL (Data Definition Language) commands in MongoDB are used to define, modify, and manage the structure of the database objects such as collections and indexes.

**1. createCollection - Create a Collection**

Creates a new collection with optional parameters such as size and validation rules.

**Example:**

db.createCollection("users", {

capped: true,

size: 5242880,

max: 5000

})

This creates a capped collection named users with a maximum size of 5MB and a limit of 5000 documents.

**2. drop - Drop a Collection**

Removes a collection and all its documents.

**Example:**

db.users.drop()

This deletes the users collection from the database.

**3. renameCollection - Rename a Collection**

Changes the name of an existing collection.

**Example:**

db.users.renameCollection("customers")

This renames the users collection to customers.

**4. createIndex - Create an Index**

Adds an index to a collection to improve query performance.

**Example:**

db.users.createIndex({ email: 1 })

This creates an ascending index on the email field.

**5. dropIndex - Drop an Index**

Removes an existing index from a collection.

**Example:**

db.users.dropIndex("email\_1")

This deletes the index on the email field.

**6. dropDatabase - Drop a Database**

Deletes the current database along with all its collections.

**Example:**

db.dropDatabase()

This removes the entire database currently in use.

**Summary Table of MongoDB DDL Commands**

| **Command** | **Description** |
| --- | --- |
| db.createCollection() | Creates a new collection |
| db.collection.drop() | Deletes a collection |
| db.collection.renameCollection() | Renames a collection |
| db.collection.createIndex() | Creates an index |
| db.collection.dropIndex() | Removes an index |
| db.dropDatabase() | Deletes the current database |

**DML (Data Manipulation Language) Commands in MongoDB**

DML commands in MongoDB are used to manipulate data within collections. These commands include **insert, update, delete, and find** operations.

**1. Insert Documents (insertOne and insertMany)**

Used to add new documents to a collection.

**Example 1: Insert a Single Document**

db.users.insertOne({

"\_id": 1,

"name": "Alice",

"age": 25,

"city": "New York"

})

**Example 2: Insert Multiple Documents**

db.users.insertMany([

{ "\_id": 2, "name": "Bob", "age": 30, "city": "Los Angeles" },

{ "\_id": 3, "name": "Charlie", "age": 35, "city": "Chicago" }

])

**2. Read Data (find, findOne)**

Used to retrieve documents from a collection.

**Example 1: Find All Documents**

db.users.find()

**Example 2: Find a Specific Document**

db.users.findOne({ "name": "Alice" })

**Example 3: Find with Condition**

db.users.find({ "age": { "$gt": 25 } })

This query retrieves users older than 25.

**3. Update Documents (updateOne, updateMany, replaceOne)**

Used to modify existing documents in a collection.

**Example 1: Update a Single Document**

db.users.updateOne(

{ "name": "Alice" },

{ "$set": { "age": 26 } }

)

This updates Alice's age to 26.

**Example 2: Update Multiple Documents**

db.users.updateMany(

{ "city": "New York" },

{ "$set": { "city": "San Francisco" } }

)

This updates the city of all users from "New York" to "San Francisco."

**Example 3: Replace a Document**

db.users.replaceOne(

{ "name": "Alice" },

{ "\_id": 1, "name": "Alice Cooper", "age": 26, "city": "New York" }

)

This completely replaces Alice’s document.

**4. Delete Documents (deleteOne, deleteMany)**

Used to remove documents from a collection.

**Example 1: Delete a Single Document**

db.users.deleteOne({ "name": "Charlie" })

Deletes the first document where the name is "Charlie."

**Example 2: Delete Multiple Documents**

db.users.deleteMany({ "age": { "$gt": 30 } })

Deletes all users older than 30.

**Summary of DML Commands in MongoDB**

| **Command** | **Description** |
| --- | --- |
| insertOne() | Inserts a single document. |
| insertMany() | Inserts multiple documents. |
| find() | Retrieves documents. |
| findOne() | Retrieves a single document. |
| updateOne() | Updates a single document. |
| updateMany() | Updates multiple documents. |
| replaceOne() | Replaces an entire document. |
| deleteOne() | Deletes a single document. |
| deleteMany() | Deletes multiple documents. |

In MongoDB, **DCL (Data Control Language)** commands are used to control access to the database by granting and revoking privileges. These commands help manage user permissions and security.

**DCL Commands in MongoDB**

MongoDB primarily provides two DCL commands:

1. **GRANT (Not directly available, but done via createUser and updateUser)**
2. **REVOKE (Not directly available, but done via revokeRolesFromUser or dropUser)**

**1. Creating a User (Granting Privileges)**

Instead of GRANT, MongoDB uses the createUser command to assign roles to a user.

**Example: Grant Read and Write Access**

use admin;

db.createUser({

user: "testUser",

pwd: "password123",

roles: [

{ role: "readWrite", db: "testDB" } // Allows read and write operations on testDB

]

});

✅ This creates a user testUser with **read and write** access to testDB.

**2. Updating a User (Modifying Privileges)**

You can modify a user's roles using updateUser.

**Example: Grant Additional Role**

use admin;

db.updateUser("testUser", {

roles: [

{ role: "readWrite", db: "testDB" },

{ role: "dbAdmin", db: "testDB" } // Added dbAdmin role

]

});

✅ Now, testUser also has **database administration** privileges.

**3. Revoking Privileges from a User**

MongoDB does not have a direct REVOKE command, but you can remove roles using updateUser or revokeRolesFromUser.

**Example: Revoke a Role**

use admin;

db.runCommand({

revokeRolesFromUser: "testUser",

roles: [{ role: "dbAdmin", db: "testDB" }]

});

✅ This removes the **dbAdmin** role from testUser.

**4. Deleting a User (Revoking All Privileges)**

To completely remove a user and revoke all their access, use dropUser.

**Example: Remove a User**

use admin;

db.dropUser("testUser");

✅ This removes testUser from the database, revoking all privileges.

**5. Listing User Privileges**

To check a user’s roles and privileges:

use admin;

db.getUser("testUser");

✅ This displays all roles assigned to testUser.

**Conclusion**

* MongoDB does **not** have direct GRANT and REVOKE commands.
* **User permissions** are controlled via createUser, updateUser, revokeRolesFromUser, and dropUser.
* Roles define access levels, such as read, readWrite, dbAdmin, etc.

Indexes in MongoDB improve the efficiency of queries by allowing the database to locate data faster. Below are different types of indexes in MongoDB with examples:

**1. Single Field Index**

* Created on a single field to improve query performance.
* Example:
* db.users.createIndex({ name: 1 }) // Ascending order
* Queries using the name field will be optimized.

**2. Compound Index**

* Created on multiple fields to support queries using both fields.
* Example:
* db.users.createIndex({ name: 1, age: -1 }) // Ascending on `name`, descending on `age`
* Optimized for queries that filter by name and age.

**3. Multikey Index**

* Created on array fields where each element is indexed.
* Example:
* db.products.createIndex({ tags: 1 })
* Queries filtering tags like { tags: "electronics" } benefit from the index.

**4. Text Index**

* Used for full-text search on string fields.
* Example:
* db.articles.createIndex({ content: "text" })
* Enables search with $text queries:
* db.articles.find({ $text: { $search: "mongodb" } })

**5. Hashed Index**

* Used for sharding and supports equality queries.
* Example:
* db.users.createIndex({ email: "hashed" })
* Optimized for exact match searches like { email: "user@example.com" }.

**6. Wildcard Index**

* Indexes all fields dynamically without explicitly specifying them.
* Example:
* db.logs.createIndex({ "$\*\*": 1 })
* Useful for queries on dynamic or unknown fields.

**7. Geospatial Index**

* Used for location-based queries.
* **2dsphere** for latitude/longitude:
* db.places.createIndex({ location: "2dsphere" })
* **2d** for flat-plane coordinates:
* db.places.createIndex({ location: "2d" })

**8. Unique Index**

* Ensures that field values are unique.
* Example:
* db.users.createIndex({ username: 1 }, { unique: true })
* Prevents duplicate username values.

**9. Partial Index**

* Indexes only documents meeting a specific condition.
* Example:
* db.orders.createIndex({ status: 1 }, { partialFilterExpression: { status: "shipped" } })
* Index only applies to orders where status is "shipped".

**10. TTL (Time-To-Live) Index**

* Automatically deletes documents after a certain period.
* Example:
* db.sessions.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 })
* Removes sessions after 1 hour.

**Choosing the Right Index:**

* Use **single field** indexes for simple queries.
* Use **compound** indexes when multiple fields are frequently queried together.
* Use **text** indexes for search functionality.
* Use **geospatial** indexes for location-based queries.
* Use **hashed** indexes for sharded collections.

**Aggregation Framework in MongoDB**

The **Aggregation Framework** in MongoDB is a powerful feature used for processing and analyzing data within collections. It allows performing complex transformations and computations on data, similar to SQL's GROUP BY and aggregation functions.

The framework operates using a **pipeline** concept, where data passes through multiple **stages**, each transforming it in some way.

**Key Stages in Aggregation Framework**

1. **$match** → Filters documents (like WHERE in SQL).
2. **$group** → Groups data by a specific field and applies aggregate functions.
3. **$project** → Modifies document structure (includes/excludes fields, computes new fields).
4. **$sort** → Sorts documents based on specified fields.
5. **$limit** → Restricts the number of documents.
6. **$skip** → Skips a specified number of documents.
7. **$unwind** → Deconstructs arrays into separate documents.
8. **$lookup** → Performs a left join with another collection.
9. **$addFields** → Adds new fields to documents.

**Example Use Cases**

**1. Filtering Data Using $match**

Filters documents where age > 25:

db.users.aggregate([

{ $match: { age: { $gt: 25 } } }

])

**2. Grouping and Counting Using $group**

Find the total number of users per country:

db.users.aggregate([

{ $group: { \_id: "$country", totalUsers: { $sum: 1 } } }

])

**Output:**

[

{ "\_id": "USA", "totalUsers": 50 },

{ "\_id": "India", "totalUsers": 70 }

]

**3. Restructuring Data Using $project**

Selecting only name and age, and renaming age to userAge:

db.users.aggregate([

{ $project: { \_id: 0, name: 1, userAge: "$age" } }

])

**4. Sorting Data Using $sort**

Sort users by age in descending order:

db.users.aggregate([

{ $sort: { age: -1 } }

])

**5. Pagination Using $skip and $limit**

Get the first 5 users after skipping the first 10:

db.users.aggregate([

{ $skip: 10 },

{ $limit: 5 }

])

**6. Unwinding Arrays Using $unwind**

If a document has an array field, $unwind creates separate documents for each array element.

db.orders.aggregate([

{ $unwind: "$items" }

])

For a document like:

{ "\_id": 1, "items": ["Apple", "Banana"] }

The result will be:

[

{ "\_id": 1, "items": "Apple" },

{ "\_id": 1, "items": "Banana" }

]

**7. Joining Collections Using $lookup**

Join users collection with orders based on userId:

db.users.aggregate([

{

$lookup: {

from: "orders",

localField: "userId",

foreignField: "userId",

as: "userOrders"

}

}

])

This will embed matching orders into the users collection.

**Conclusion**

The **Aggregation Framework** in MongoDB is a powerful tool for data transformation, analysis, and reporting. By leveraging different stages, you can perform complex queries efficiently without requiring multiple database calls.

**Normalization in MongoDB**

Normalization in MongoDB is the process of structuring data to minimize data redundancy and improve data integrity. It follows principles similar to relational databases but differs in implementation due to MongoDB’s flexible schema and document-based storage model.

**Normalization vs. Denormalization in MongoDB**

* **Normalization** involves breaking data into multiple collections and using references.
* **Denormalization** involves embedding related data within a single document.

**Example of Normalization in MongoDB**

**Scenario: A Library System**

Imagine a system that stores **books** and **authors**.

**Normalized Structure (Using References)**

We store **authors** and **books** in separate collections and use references to link them.

**Author Collection**

{

"\_id": ObjectId("64c9a1f8a1b23e5d6b9e1a11"),

"name": "J.K. Rowling",

"country": "UK"

}

**Book Collection**

{

"\_id": ObjectId("64c9a2f9b2c34e7d7c9f2b22"),

"title": "Harry Potter and the Sorcerer's Stone",

"author\_id": ObjectId("64c9a1f8a1b23e5d6b9e1a11"),

"published\_year": 1997

}

**Query to Fetch Book with Author Information**

db.books.aggregate([

{

$lookup: {

from: "authors",

localField: "author\_id",

foreignField: "\_id",

as: "author\_details"

}

}

])

**Advantages of Normalization**

* Reduces redundancy (author details stored once).
* Smaller document sizes lead to better performance for some queries.
* Easier to maintain consistency (update author details in one place).

**Example of Denormalization in MongoDB**

Instead of references, we embed author details directly within the book document.

**Denormalized Book Document**

{

"\_id": ObjectId("64c9a2f9b2c34e7d7c9f2b22"),

"title": "Harry Potter and the Sorcerer's Stone",

"author": {

"name": "J.K. Rowling",

"country": "UK"

},

"published\_year": 1997

}

**Advantages of Denormalization**

* Faster reads (no need for $lookup).
* Simpler queries since data is self-contained.

**Disadvantages**

* Redundant data (if an author writes multiple books, their details are repeated).
* Updating author details requires updating multiple documents.

**When to Use Normalization in MongoDB?**

* When data changes frequently (e.g., updating author details).
* When documents would grow too large if embedded.
* When relationships are complex and involve multiple entities.

**Conclusion**

Normalization in MongoDB is useful for reducing redundancy and maintaining consistency. However, MongoDB’s document model often favors denormalization for performance reasons, especially in read-heavy applications. Choosing between normalization and denormalization depends on the specific use case and performance needs.