

MongoDB

<https://www.mongodb.com/docs/manual/>

1. What is MongoDB and what are the features of MongoDB? How is it different from traditional relational databases?

MongoDB is a document-oriented, NoSQL database management system that stores data in flexible, JSON-like documents known as BSON (Binary JSON), enabling a more dynamic and scalable approach to data storage. It is designed to be schema-less, providing flexibility in handling unstructured or semi-structured data.

Features of MongoDB:

- **Document-Oriented:** Stores data in BSON documents.
- **Schema-less:** No predefined structure required for data insertion.
- **Dynamic Schema:** Supports varying field structures in documents.
- **Indexing:** Enhances query performance through indexing.
- **Query Language:** Supports a rich set of queries, including field, range, and regex searches.
- **Aggregation Framework:** Allows complex data processing and transformation.
- **Horizontal Scalability:** MongoDB is designed for horizontal scalability, allowing us to scale out our database by adding more servers to our MongoDB cluster.
- **Replication:** MongoDB supports replica sets, providing high availability and fault tolerance by maintaining multiple copies of data across different servers.
- **Sharding:** MongoDB supports sharding, which allows distributing data across multiple servers to improve performance and manage large datasets.
- **Geospatial Indexing:** MongoDB has built-in support for geospatial indexing and queries, making it suitable for location-based applications.

Differences from Traditional Relational Databases:

- **Schema:** MongoDB is schema-less; relational databases have a fixed schema.
- **Data Model:** MongoDB uses a document-oriented model; relational databases use tables.
- **Scalability:** MongoDB scales horizontally; relational databases often scale vertically.
- **Join Operations:** MongoDB doesn't support traditional joins; relational databases rely on them.
- **Use Cases:** MongoDB is suitable for unstructured data; relational databases excel in scenarios requiring a rigid schema and complex transactions.

To download MongoDB:

- I. MongoDB Community Server Download: <https://www.mongodb.com/try/download/community>
- II. MongoDB Shell Download: <https://www.mongodb.com/try/download/shell>
- III. MongoDB Command Line Database Tools Download: <https://www.mongodb.com/try/download/database-tools>

Note: To check MongoDB version: using cd\ go to › C:\Program Files\MongoDB\Server\7.0\bin> mongod --version↵
To check the MongoDB Shell (mongosh): › C:\Program Files\MongoDB\Server\7.0\bin> mongosh ↵

Commands:

- show dbs; / show databases: Lists all available databases.
- use <database-name>; Switches to the specified database or **creates a new one**.
- db.dropDatabase(); Deletes the currently selected database.
- show collections;; Lists collections in the current database.
- cls;; Clears the Command Prompt or terminal screen.
- db.createCollection('<collection-name>'); Creates a new collection in the current database.
- db.<collection-name>.drop(); Deletes the specified collection in the current database.

MongoDB

<https://www.mongodb.com/docs/manual/>

Start Cmd: ▶ C:\Program Files\MongoDB\Server\7.0\bin> mongosh ◀

test> use Employee; # Switch to the "Employee" database

switched to db Employee

Employee> show collections; # Show collections in the current database

Employee> db.createCollection('data'); # Create a collection named 'data' in the "Employee" database

{ ok: 1 }

Employee> show collections; # Show collections after the creation

Data

Employee> show databases; # Show databases with their sizes

Employee 8.00 KiB

admin 40.00 KiB

config 108.00 KiB

local 40.00 KiB

Employee> show collections;

Data

Employee> db.data.drop(); # Drop the 'data' collection from the "Employee" database

True

Employee> show.collections; # Show collections after dropping 'data' (should be empty)

Employee> db.dropDatabase(); # Drop the entire "Employee" database

{ ok: 1, dropped: 'Employee' }

Employee> show dbs; # Show databases after dropping "Employee" (it should not be present)

admin 40.00 KiB

config 108.00 KiB

local 40.00 KiB

Employee>

Insert a single document into the 'data' collection in the "Employee" database

Employee> db.data.insertOne({'name':'Abhinas', age: 25});

Output: {

acknowledged: true,

insertedId: ObjectId('658d3dd93478002c5d44bdb9')

}

Insert multiple documents into the 'data' collection in the "Employee" database

Employee> db.data.insertMany([

{'name':'Abhishek', age: 26},

{'name':'Rohit', age: 21},

{'name':'Anuj', age: 24}

]);

Find and display all documents in the 'data' collection in the "Employee" database

Employee> db.data.find();

• Explain the BSON data format used in MongoDB.

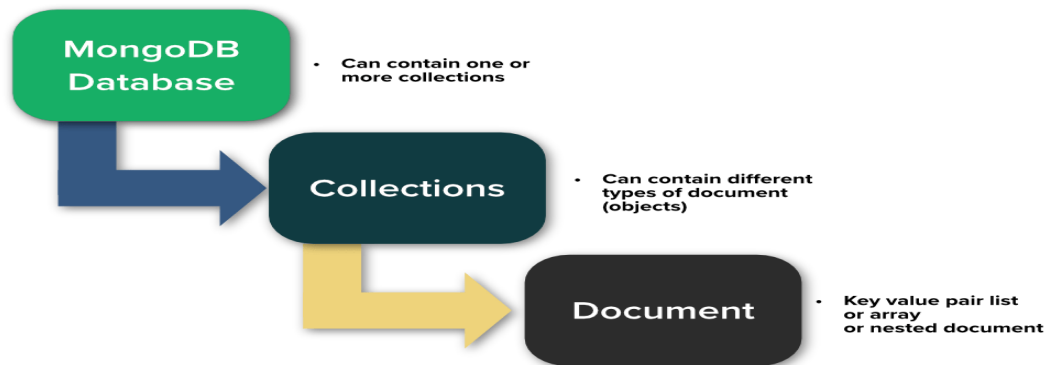
BSON is a binary-encoded data format used by MongoDB to represent and store data in a compact and efficient manner.

- **Representation:** BSON utilizes a binary format for efficiency in storage and transmission.
- **Data Types:** It supports a variety of data types, encompassing strings, numbers, arrays, and more.
- **Efficiency:** BSON's compact encoding enhances efficiency in both storage and transmission.
- **MongoDB Storage:** BSON is actively used for storing data within the MongoDB database.
- **Flexibility:** BSON provides flexibility by allowing MongoDB to seamlessly adapt to new data types.

- **What is a document in MongoDB?**

In MongoDB, a document is a JSON-like data structure used for storing and organizing data. It consists of key-value pairs, where keys are strings and values can be various data types, including other documents, arrays, strings, numbers, and booleans. Documents are stored in collections, which are similar to tables in relational databases. Each document has a unique identifier (`_id`) within its collection. MongoDB's flexible schema allows documents in the same collection to have different fields, making it suitable for handling diverse data structures.

- **Differentiate between a collection and a database in MongoDB.**



1. A database is a container for collections, and collections are containers for documents.
2. Databases help organize and manage collections, while collections store individual documents with flexible and schema-less structures.
3. A database is a physical container for collections, while a collection is a group of MongoDB documents.
4. A database is equivalent to a relational database, while a collection is similar to a table in a relational database.
5. MongoDB can have multiple databases, each with its own collections and documents, while collections do not enforce a schema, allowing documents within a collection to have different structures.
6. Databases are used to organize and store collections, while documents in a collection are JSON-like objects with key-value pairs.

- **How does MongoDB provide horizontal scalability?**

In MongoDB, Sharding and replication are two common strategies used to achieve horizontal scaling.

Sharding is focused on increasing the capacity of the system by distributing data, while replication is focused on enhancing availability and reliability by maintaining redundant copies of the data. Both strategies are often used together to achieve a balance of increased capacity, fault tolerance, and high availability in horizontally scalable systems.

1. **Sharding:**

- Sharding involves breaking a large database into smaller, more manageable parts called shards.
- Each shard is hosted on a separate machine or node, distributing the data and workload across multiple servers.
- Sharding primarily aims to increase the overall capacity of the system by allowing it to handle larger datasets and increased transactional throughput.

2. **Replication:**

- Replication involves creating and maintaining copies of the data on multiple servers.
- Each copy of the data is known as a replica, and it provides redundancy and fault tolerance.

MongoDB

<https://www.mongodb.com/docs/manual/>

- Replication enhances the availability and reliability of the system by ensuring that if one server fails, another replica can take over, preventing data loss and minimizing downtime.
- **Ordered and Unordered inserts:** In MongoDB, ordered insert and unordered insert refer to the behavior of bulk write operations, such as **insertMany()**, where multiple documents are inserted into a collection in a single operation.

Ordered Insert: In an ordered insert, MongoDB processes the documents in the order they are provided in the array. If an error occurs during the insertion of a document, MongoDB will stop processing further documents in the array.

Example:

```
db.collection.insertMany([
  { _id: 1, name: "Alice" },
  { _id: 2, name: "Bob" },
  { _id: 3, name: "Charlie" },
],
{ ordered: true }
);
```

Unordered Insert: In an unordered insert, MongoDB continues to process the remaining documents in the array even if an error occurs during the insertion of a particular document. It doesn't stop the insertion process due to errors.

Example:

```
db.collection.insertMany([
  { _id: 1, name: "Alice" },
  { _id: 2, name: "Bob" },
  { _id: 3, name: "Charlie" },
],
{ ordered: false }
);
```

- **Find:** `db.data.find({'Key':'value'});` → `db.data.find({'name':'Abhishek'});`
- **findOne:** Find documents where the field named "name" has the value 'Abhishek'
`db.data.findOne({'Key':'value'});` → `db.data.find({'name':'Abhishek'});`
- **Projection** is used to specify which fields should be included or excluded in the result set. Example:
→ `db.collection.find({}).project({ name: 1, age: 1, _id: 0 });` This returns documents with only the "name" and "age" fields, excluding the "_id" field.
- **The sort** method is used to sort the result set based on one or more fields. Example:
→ `db.collection.find({}).sort({ age: 1 });` This sorts the result set in ascending order based on the "age" field.
- **Limit and Skip:** The limit method restricts the number of documents returned, and the skip method skips a specified number of documents. Example: → `db.collection.find({}).limit(10).skip(5);` This returns 10 documents, starting from the 6th document in the result set.
- **Aggregation:** The aggregation framework provides powerful tools for data transformation and processing.
Example: → `db.collection.aggregate([
 { $match: { status: "active" } },
 { $group: { _id: "$type", total: { $sum: "$quantity" } } },
]);` This performs an aggregation to find the total quantity for each document type with an "active" status.
- **Indexing:** Proper indexing can significantly improve the performance of read operations.

MongoDB

<https://www.mongodb.com/docs/manual/>

Example: `→ db.collection.createIndex({ name: 1 });` This creates an index on the "name" field to speed up queries that involve searching or sorting by name.

- **Comparison operators in MongoDB:**

- **Operators:**

1. `$eq`: (equal to) → Example: `{ field: { $eq: value } }`
2. `$ne`: (not equal to) → Example: `{ field: { $ne: value } }`
3. `$gt`: (greater than) → Example: `{ field: { $gt: value } }`
4. `$lt`: (less than) → Example: `{ field: { $lt: value } }`
5. `$gte`: (greater than or equal to) → Example: `{ field: { $gte: value } }`
6. `$lte`: (less than or equal to) → Example: `{ field: { $lte: value } }`
7. `$in`: (any of the values specified in an array) → Example: `{ field: { $in: [value1, value2, ...] } }`
8. `$nin`: (values that are not in the specified array) → Example: `{ field: { $nin: [value1, value2, ...] } }`
9. `$exists`: (Matches documents that have the specified field) → Example: `{ field: { $exists: true/false } }`
10. `$type`: Matches documents where the value of a field is of a specific BSON data type → `{ field: { $type: "string" } }`
11. `$regex`: Matches documents where a field matches a specified regular expression. → `{ field: { $regex: /pattern/ } }`
 - `customers> db.cusInfo.find({'_id':'017'});`
 - `customers> db.cusInfo.find({'_id':{'$gte':'017'}});`
 - `customers> db.cusInfo.find({'_id':{'$gte':'017'}}).count();`
 - `customers> db.cusInfo.insertMany([{'_id':'023','name':'RaviRanjan'},{'_id':'024','name':'Juhi'}]);`
 - `customers> db.cusInfo.insert({'_id':'028','name':'AbhihekAbhi'});`
 - `customers> db.cusInfo.find({'_id':{'$in':['017','018','021']}});`
 - `customers> db.cusInfo.find({'_id':{'$nin':['016','018','021']}});`

- **Cursor:** In MongoDB, when the `find()` method is used to find the documents present in the given collection, then this method returned a pointer which will points to the documents of the collection, now this pointer is known as cursor. Or in other words we can say that a *cursor is a pointer*, and using this pointer we can access the document. By default, cursor iterate automatically, but we can iterate a cursor manually. It does not immediately fetch all the documents; instead, it allows us to iterate over the result set, fetching documents as needed.

Example1:

```
var data = db.cusInfo.find({'_id': { $in: ['017', '018', '021'] } });
    data.forEach(function(doc) {
        printjson(doc);
    });
```

Example2:

```
var data = db.cusInfo.find(); // Querying all documents in the 'cusInfo' collection
while (data.hasNext()) {    // Iterating over the cursor manually
    var doc = data.next();   // Retrieving the next document in the result set
    printjson(doc);         // Printing the document in JSON format
}
```

Cursor methods:

- **limit()**: Limits the number of documents returned by the cursor.
Example: `db.collection.find().limit(5);`
- **skip()**: Skips a specified number of documents from the beginning of the cursor.
Example: `db.collection.find().skip(10);`
- **sort()**: Sorts the documents in the cursor based on specified criteria. We use 1 Ascending order and -1 for descending.
Example: `customers> db.cusInfo.find({'_id':{'$gt':'010'}}).sort({'_id':-1});` //Descending order sorting
- **count()**: Returns the count of documents in the cursor without fetching them.
Example: `db.collection.find().count();`
- **toArray()**: Converts the cursor into an array of documents.

```
var cursor = db.collection.find();
var documentArray = cursor.toArray();
```

- **close():** Closes the cursor. It is usually done implicitly, but you can manually close it if needed.
var cursor = db.collection.find();
cursor.close();

- **Note:→ Insertion:** db.products.insertMany([{'name':'Laptop','price':55078,'address':'Delhi'}, {'name':'CPU','price':37044,'address':'Bangalore'}]);

Example: customers> db.products.find({'price':{\$gt:500}}).limit(5).sort({'price':-1});

▪ Logical operators:

- **\$and:** Joins multiple query expressions and returns documents that match all of the conditions.
Syntax: db.collection.find({ \$and: [{ condition1 }, { condition2 }] });
Example: db.products.find({\$and:[{'price':{\$gt:2000}},{'address':{\$eq:'Patna'}}]});
- **\$or:** Joins multiple query expressions and returns documents that match at least one of the conditions.
Syntax: db.collection.find({ \$or: [{ condition1 }, { condition2 }] });
Example: db.products.find({\$or:[{'price':{\$gt:200000}},{'address':{\$eq:'Delhi'}}]});
- **\$not:** Inverts the effect of a query expression and returns documents that do not match the specified condition.
Syntax: db.collection.find({ field: { \$not: { \$eq: value } } });
Example: db.products.find({'price':{\$not:\$eq:21034}}); **Or we can also use:** db.products.find({'price':{\$ne:21034}});
- **\$nor:** Joins multiple query expressions with the **\$nor** operator and returns documents that do not match any of the conditions.
Syntax: db.collection.find({ \$nor: [{ condition1 }, { condition2 }] });
Example: db.products.find({\$nor:[{'price':{\$gt:200000}},{'address':{\$eq:'Noida'}}]});

- **Complex expression:** In MongoDB, a complex expression is one that involves more than basic comparisons or simple field values. MongoDB provides a rich set of query operators and expressions to handle complex queries. The \$expr operator is specifically designed for building complex query expressions, especially when we need to compare fields within the same document or perform advanced logical operations.

Moreover, the \$expr operator is commonly used in conjunction with the aggregation framework operators, allowing us to leverage the powerful aggregation capabilities of MongoDB within regular queries. By using these tools together, we gain flexibility and expressiveness when querying and filtering data based on a variety of conditions and calculations.

For example, if we want to find all products in our 'products' collection where the price is greater than 50000, we can use the following query: ➔ db.products.find({ \$expr: { \$gt: ['\$price', 50000] } });

This query will return all documents in the 'products' collection where the value of the 'price' field is greater than 50000.

- Element operators in MongoDB are used to query data based on field existence or data types. There are two element operators in MongoDB: \$exists and \$type.

Elements:

- **Field:** A field is a key-value pair in a MongoDB document. Each field contains a specific piece of data.
- **Value:** The data associated with a field. Values can be of various types, including strings, numbers, arrays, documents, etc.
- **Document:** A document is a set of key-value pairs. In MongoDB, a collection is made up of documents.

- **\$exists:** This operator matches documents that have the specified field. It takes a boolean value of true or false as its argument. For example, the following query will return all documents that have a field named field1: Syntax: db.collection.find({field1: {\$exists: true}});

MongoDB

<https://www.mongodb.com/docs/manual/>

Example: `db.products.find({price:{$exists:true}});` This will find all documents in the products collection that have a price field. The `$exists` operator matches documents that contain the field, including documents where the field value is null.

- **\$type:** This operator selects documents if a field is of the specified type. The following table lists the available types and their corresponding integer values:

1.	Double	1	
2.	String	2	<code>db.products.find({name:{\$type:'string'}});</code>
3.	Object	3	
4.	Array	4	
5.	Binary data	5	
6.	Undefined	6	
7.	Object id	7	
8.	Boolean	8	<code>'bool'</code>
9.	Date	9	
10.	Null	10	
11.	Regular Expression	11	
12.	JavaScript	13	
13.	Symbol	14	
14.	JavaScript (with scope)	15	
15.	32-bit integer	16	<code>'number' → db.products.find({price:{\$type:'number'}});</code>
16.	Timestamp	17	
17.	64-bit integer	18	
18.	Decimal128	19	
19.	Min key	-1	
20.	Max key	127	

Syntax: `db.collection.find({field1: {$type: 2}});` Example: `customers> db.products.find({price:{$type:16}});`

- **\$size:** The `$size` operator is used to get the number of elements in an array field. It matches any array with the number of elements specified by the argument.

Example: Suppose we have a collection of documents representing orders, and each document has an array field called "items" that contains the ordered items. We want to find orders with a specific number of items, let's say 3. The query would look like this: `→ db.orders.find({ items: { $size: 3 } });`

- **Projection:** In MongoDB, projection refers to the mechanism of selecting only the specific fields from a document that you want to retrieve in a query result. It allows you to limit the amount of data transferred over the network and can improve query performance by retrieving only the necessary fields.

Example: Now, let's say you want to retrieve only the **name** and **email** fields from the documents:
`db.users.find({}, { _id: 0, name: 1, email: 1 })` // Projection to include only 'name' and 'email'

Note: it's important to note that including and excluding fields in the same projection is not allowed. We can either include fields or exclude fields, but not both simultaneously.

- **Note:** The `insertOne` operation is inserting a document into the **AllShop** collection. The document has a field named 'CPU', which is an array containing objects representing CPUs. Each CPU object has fields like 'name', 'price', and 'address'.

```
shop> db.AllShop.insertOne({
  'CPU': [
    {'name': 'SAMSUNG', 'price': 36739, 'address': 'Patna'},
    {'name': 'HP', 'price': 4564, 'address': 'Jaipur'}
  ]
});
```

The **find** operation is attempting to retrieve documents from the **AllShop** collection where the name of a CPU within the 'CPU' array is equal to 'SAMSUNG'. `→shop> db.AllShop.find({'CPU.name': 'SAMSUNG'})`
`→ shop> db.AllShop.find({'CPU.price':{$gt:31331}});`

MongoDB

<https://www.mongodb.com/docs/manual/>

- **In MongoDB, the \$all operator** is used in queries to match documents that contain an array field with all of the specified values. It checks whether the array field contains all the values specified in the query. Syntax for using **\$all**: **db.collection.find({ arrayField: { \$all: [value1, value2, ...] } })**

Example: `shop> db.AllShop.find({'CPU.name':{$all:['SAMSUNG','HP']}});`

- **\$elemMatch** is a query operator in MongoDB that is used to match documents that contain an array field with at least one element that matches all the specified query criteria. The \$elemMatch operator is used to find documents with at least one array field. The finding operation matches the data in the array field with the criteria mentioned with the \$elemMatch.

Example: Let's assume we have a collection called users, and each document in the collection has an array field called orders. Each order is represented as an embedded document with product and quantity fields. Now, let's say we want to find users who have placed an order for at least 2 laptops. We can use \$elemMatch for this:

```
db.users.find({
  "orders": { // Look into the 'orders' array within each document
    $elemMatch: { // Apply $elemMatch to ensure both conditions are satisfied within the same array element
      "product": "Laptop",
      "quantity": { $gte: 2 }
    }
  }
})
```

- In MongoDB, update operations are used to modify documents in a collection. MongoDB provides various update operators to perform different types of updates.

MongoDB provides several update operators that can be used to modify the data in a collection. Some of the commonly used update operators are:

\$set: Sets the value of a field in a document.

\$unset: Removes the specified field from a document.

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

\$push: Adds an item to an array.

\$pull: Removes all array elements that match a specified query.

- **Update:** `customers> db.cusInfo.update({'_id':'012'},{$set:{'name':'Ashok'}});`
- **UpdateOne:** `customers> db.cusInfo.updateOne({'_id':'018'},{$set:{'name':'Anuj'}});`
- **UpdateMany:** The updateMany method in MongoDB is used to update multiple documents in a collection that match the specified filter criteria. Example: `db.myCollection.updateMany(
 { 'status': 'Pending' }, // Update documents where status is 'Pending'
 { $set: { 'status': 'Processed' } } // Set status to 'Processed'
);`
- Increment the "age" field by 1 for the document with "username" equal to "john_doe"
`db.users.update(
 { "username": "john_doe" },
 { $inc: { "age": 1 } }
)`
- The **\$push** operator is used to add an element to an array.
`db.collection.update(// Add the value "New Task" to the "tasks" array for the document with
 "_id" equal to ObjectId("...")
 { "_id": ObjectId("...") },
 { $push: { "tasks": "New Task" } })`

MongoDB

<https://www.mongodb.com/docs/manual/>

- **Removing a Field:** The \$unset operator is used to remove a field from a document. Example:
customers> db.cusInfo.update({'_id':'023'},{\$unset:{'name':''}}); In this example, the name will be removed from the document with the specified _id.

Note for set we can use: customers> db.cusInfo.update({'_id':'023'},{\$set:{'name':'Arun'}});

- **Renaming a Field:** The \$rename operator is used to rename a field in a document.
customers> db.cusInfo.update({'_id':'023'},{\$rename:{'name':'cusName'}});

Note: we can also use \$unset and \$rename with updateMany query.

- The \$push operator is used in update operations to add elements to an array field. It is particularly useful when you want to append one or more values to an existing array without modifying the other elements in the array or the document itself.

Example: Create a collection and insert data in array form→

```
customers> db.createCollection('arrDoc');
```

```
{ ok: 1 }
```

- customers> db.arrDoc.insertOne({'Fruits':['Mango','Banana','Apple','Orange']});
- customers>db.arrDoc.insertMany([{'company':['Oracle','Infosys','HCL','TATA']},{ 'jobType':['Manager','Developer','Human Resource','Technical support','Project management']}]);

// Pushing 'Grapes' to the 'Fruits' array:

```
db.arrDoc.update( { "_id": ObjectId('659e2f8986761cd9facedfb') }, { "$push": { "Fruits": "Grapes" } } );
```

// Pushing multiple fruits to the 'Fruits' array

```
db.arrDoc.update( { "_id": ObjectId('659e2f8986761cd9facedfb') },  
{ "$push": { "Fruits": { "$each": ["Grapes", "Watermelon", "Pineapple"] } } } );
```

The \$pop method is used to remove elements from the end of an array. There are two values with can use with \$pop:

- To remove the last element from an array, we can use **1**.
- To remove the first element from an array, we can use **-1**.

// Remove the last element from the 'Fruits' array

```
db.arrDoc.update( { "_id": ObjectId('659e2f8986761cd9facedfb') }, { "$pop": { "Fruits": 1 } } );
```

- If we want to remove a specific element from an array we can use the \$pull operator.
Example: customers> db.arrDoc.update({'_id': ObjectId('659e2f8986761cd9facedfb')},{\$pull:{'Fruits':'Banana'}});
- The delete operation typically refers to the removal of documents from a collection. There are several ways to perform delete operations in MongoDB, and the choice depends on the specific requirements and use case.
 - **deleteOne:** Deletes a single document that matches the specified criteria.
Example: db.users.deleteOne({ username: "john_doe" });
 - **deleteMany:** Deletes all documents that match the specified criteria.
Example: db.products.deleteMany({ stock_quantity: { \$lte: 0 } });
 - db.orders.remove({ status: "cancelled" }); This removes all documents in the 'orders' collection where the 'status' is "cancelled".
 - **Drop Collection:** Removes an entire collection, including all of its documents. Example: db.collection.drop();

- **Explain the concept of indexing in MongoDB and its importance.**

Indexing in MongoDB is a process of creating data structures that improve the speed of data retrieval operations on a MongoDB database. Indexes provide a quick and efficient way to locate documents in a collection based on the values of certain fields. By creating indexes on frequently queried fields, MongoDB can significantly enhance query performance.

Importance of Indexing in MongoDB:

1. **Improved Query Performance:** Speeds up queries by providing a quick path to relevant documents.
2. **Efficient Sorting:** Facilitates efficient sorting operations.
3. **Enhanced Aggregation:** Improves performance for aggregation queries.
4. **Accelerated Range Queries:** Efficient for range queries (e.g., **\$gt**, **\$lt**).
5. **Unique Constraints:** Enforces uniqueness constraints on fields.
6. **Text Search and Geospatial Queries:** Supports specialized index types for these queries.

Index Structure: MongoDB uses B-tree data structures for indexing. These structures allow for efficient searching, insertion, and deletion of keys. Each index consists of a set of keys derived from the values of one or more fields in the documents.

❖ Types of Indexes:

Single Field Index: `db.collection.createIndex({ field: 1 });` // 1 for ascending, -1 for descending

Compound Index: `db.collection.createIndex({ field1: 1, field2: -1 });`

Multikey Index: Used for arrays. `db.collection.createIndex({ "arrayField": 1 });`

Text Index: Used for text search. `db.collection.createIndex({ textField: "text" });`

→ **db.cusInfo.getIndexes();** Shows existing indexes on **cusInfo** collection.

→ **db.cusInfo.dropIndex({'name':1});** Deletes the index on the 'name' field in **cusInfo** collection. Use with caution; it affects query performance.

→ **Simple explain:** Returns information about the query execution plan, including details about index usage, number of scanned documents, and other relevant statistics.

Example: `db.collection.find({ field: "value" }).explain();`

→ **explain with executionStats:** Provides more detailed execution statistics, including time taken for different stages of query execution.

Example: `db.collection.find({ field: "value" }).explain("executionStats");`

- **Aggregation:** In MongoDB, aggregation is a powerful framework that allows us to process and transform data from a collection and return computed results. Aggregation is often used for tasks such as grouping, filtering, and performing calculations on data.

The aggregation pipeline is a framework that allows users to perform complex data processing operations on MongoDB collections. The pipeline consists of a sequence of stages, each of which performs a specific operation on the data. The output of one stage is passed as input to the next stage, allowing users to build up complex queries.

The aggregation framework is versatile and can handle a wide range of data processing tasks, making it a valuable tool for extracting meaningful insights from MongoDB collections.

- **\$group:** Used for grouping documents based on specified criteria and performing aggregate operations on each group. Example:

```
db.sales.aggregate([
  {
    $group: {
      _id: "$product",
      totalQuantity: { $sum: "$quantity" },
      averagePrice: { $avg: "$price" }
    }
  }
])
```

Explanation:

- **\$group** is the first stage in the aggregation pipeline.
- **_id: "\$product"**: Groups documents by the "product" field, meaning it will create separate groups for each distinct product.
- **totalQuantity: { \$sum: "\$quantity" }**: Calculates the total quantity sold for each product by summing up the "quantity" field within each group.
- **averagePrice: { \$avg: "\$price" }**: Calculates the average price for each product by averaging the "price" field within each group.

MongoDB

<https://www.mongodb.com/docs/manual/>

- **\$match:** Used for filtering documents based on specified criteria, allowing us to include or exclude documents from further processing in the aggregation pipeline. Example:

```
db.sales.aggregate([
  {
    $match: {
      product: "A"
    }
  }
])
```

Explanation:

- **db.sales.aggregate([...]):** Initiates the MongoDB aggregation pipeline on the "sales" collection.
- **\$match:** The first and only stage in the pipeline, used for filtering documents based on specified criteria.
- **{ product: "A" }:** Specifies the condition for filtering. This stage ensures that only documents with the "product" field equal to "A" will be included in the result.

Example: Query 1: Find products with the name 'Pendrive'

→ customers> db.products.aggregate([{\$match:{'name':'Pendrive'}}]);

Query 2: Find products with a price greater than 100,000

→ customers> db.products.aggregate([{\$match:{'price':{\$gt:100000}}]);

Query 3: Count the total number of products grouped by name

→ customers> db.products.aggregate([{\$group:{_id:'\$name',totalProducts:{\$sum:1}}]])

The \$group stage groups documents by the "name" field.

The \$sum: 1 expression counts the number of documents in each group, representing the total products with that name.

This query is designed to provide a count of products grouped by their names in the "products" collection.

Query4: Aggregation query to group by product name and calculate total price

→ customers> db.products.aggregate([{\$group:{_id:'\$name',totalProducts:{\$sum:'\$price'}}]])

Query5: What is the total revenue for the "Electronics" category in the "sales" collection for the year 2023?

```
db.sales.aggregate([ // MongoDB Aggregation Query
{
  $match: { // Stage 1: Match documents for the "Electronics" category and the year 2023
    category: "Electronics",
    date: {
      $gte: ISODate("2023-01-01T00:00:00Z"),
      $lt: ISODate("2024-01-01T00:00:00Z")
    }
  },
  $group: { // Stage 2: Group by the "category" and calculate total revenue
    _id: "$category",
    totalRevenue: { $sum: { $multiply: ["$quantity", "$price"] } }
  }
}
])
```

- **\$project:** \$project is an aggregation pipeline stage in MongoDB that allows us to reshape documents by specifying the inclusion or exclusion of fields, the addition of new fields, or the resetting of the values of existing fields.

Example:

```
db.cusInfo.aggregate([
{
  $project: {
    name: 0, // Exclude the 'name' field
    _id: 1 // Include the '_id' field
  }
}
])
```

This aggregation pipeline will exclude the "name" field and include the "_id" field in the output documents.

- **\$unwind:** The \$unwind operator is used in the aggregation framework to deconstruct arrays within documents, creating a separate document for each element of the array. This is particularly useful when dealing with documents that contain arrays, and we want to perform operations on each element individually.

```
{
  "_id": 1,
  "name": "John Doe",
}
```

```
"orders": [
  { "product": "A", "quantity": 2 },
  { "product": "B", "quantity": 1 },
  { "product": "C", "quantity": 3 }
]
```

In the above example, the "orders" field is an array. If we want to perform further operations on each order separately, we can use **\$unwind**:

```
db.collection.aggregate([
  {
    $unwind: "$orders"
  }
]);
```

After applying \$unwind, the document will be transformed into three separate documents, one for each order:

```
{
  "_id": 1,
  "name": "John Doe",
  "orders": { "product": "A", "quantity": 2 }
}
{
  "_id": 1,
  "name": "John Doe",
  "orders": { "product": "A", "quantity": 2 }
}
{
  "_id": 1,
  "name": "John Doe",
  "orders": { "product": "C", "quantity": 3 }
}
```

- **\$addToSet**: The \$addToSet is an array update operator in MongoDB that adds a value to an array field only if the value is not already present in the array. If the value is already present, \$addToSet does nothing.

Suppose we have a document in a collection representing a user with an array of favorite colors:

```
{
  "_id": 1,
  "username": "john_doe",
  "favoriteColors": ["blue", "green"]
}
```

If we want to add a new favorite color to the array, but we want to ensure that the color is not already present, we can use

\$addToSet:

```
db.users.update(
  { "_id": 1 },
  { $addToSet: { "favoriteColors": "red" } }
);
```

After this update operation, if "red" was not already in the "favoriteColors" array, it will be added. If "red" was already present, the array remains unchanged.

- The **\$size** operator is used in the aggregation framework to determine the number of elements in an array. It returns the size of the array, allowing us to perform operations based on the length of an array field within documents. Example:

Suppose we have a collection of documents representing users with an array of hobbies:

```
{
  "_id": 1,
  "username": "john_doe",
  "hobbies": ["reading", "traveling", "photography"]
}
```

If we want to find users whose number of hobbies is greater than a certain value, we can use the \$size operator in the aggregation framework

```
db.users.aggregate([
  {
    $match: {
      $expr: { $gt: [{ $size: "$hobbies" }, 2] } // Find users with more than 2 hobbies
    }
  }
]);
```

- The **\$limit** stage is used in the aggregation framework to restrict the number of documents passed to the next stage in the pipeline. It limits the number of documents that reach the subsequent stages of the aggregation pipeline. Example: `db.orders.aggregate([{ $limit: 2 }])`

MongoDB

<https://www.mongodb.com/docs/manual/>

- The **\$skip** stage is used in the aggregation framework to skip a specified number of documents and pass the remaining documents to the next stage in the pipeline. It is often used in conjunction with **\$limit** to implement pagination or to skip a certain number of initial documents.
Example: `db.orders.aggregate([{ $skip: 2 }])`

- The **\$filter** stage is used in the aggregation framework to filter the elements of an array based on specified conditions. It's particularly useful when we want to filter an array field within a document and include only the elements that meet certain criteria. Example:

Consider a collection named `students` with the following documents:

```
{
  "_id": 1,
  "name": "Alice",
  "grades": [95, 80, 92, 88, 97]
}
{
  "_id": 2,
  "name": "Bob",
  "grades": [75, 88, 94, 60, 85]
}
```

Suppose we want to filter and retrieve only the grades greater than 90 for each student. We can use the **\$filter** stage in the aggregation pipeline:


```
db.students.aggregate([
  {
    $project: {
      name: 1,
      highGrades: {
        $filter: {
          input: "$grades",
          as: "grade",
          cond: { $gt: ["$$grade", 90] }
        }
      }
    }
  }
])
```

- **MongoDB Atlas:** MongoDB Atlas is a fully managed cloud database service provided by MongoDB for hosting and deploying MongoDB databases on popular cloud platforms like AWS, Azure, and Google Cloud. It simplifies the database administration tasks, offering features such as automated backups, scaling, monitoring, and security measures.
 1. Create an MongoDB Atlas Account and Sign up: <https://account.mongodb.com/account/login>
 2. Go to **Database Access** and add new database user, in **Built-in Role** select **Atlas admin role**.
 3. Go to Network Access Add IP Address Choose ALLOW ACCESS FROM ANYWHERE • CONFIRM
 4. On top go to **ANKIT'S ORG - 2024-1-23 ATLAS** Click view all organisations
 5. Create New Organization → Next → Create Organisation → New project project name → Create project
 6. Create a deployment → Choose free cloud environment → Choose provider → Take a cluster name → Create
 7. Choose below **My Local Environment** → **Add My Current IP Address** → **Finish and Close**
 8. **DEPLOYMENT** → Database, Network Access → Check status (Status should be active)
 9. Database → View Monitoring → Overview, within **REGION** Mumbai (ap-south-1) section we can see clusters.
 1. Overview → connect → Shell → **I have the MongoDB Shell installed** → Copy **connection string** for our application. → Open cmd → Paste link → Enter password → **Connected** → **show dbs**

- **MongoDB compass Download:** → <https://www.mongodb.com/try/download/compass>

Select **MongoDB Compass Download (GUI)** → In platform select **Windows 64-bit (10+)(MSI)** → **Download** → **Install**

Open MongoDB Compass → Then again Go to MongoDB Atlas → Overview → Connect → Compass → I have MongoDB Compass installed → Copy the connection string, then open MongoDB Compass → In URI Section clear it and paste the copied link, (Note: within link there is password section remove it with angle bracket and there write correct password) → Connect

➤ In MongoDB Compass Create Database from  icon with database name and collection name → Here we can easily import our Created BSON file.

➤ Now in **shell** we can also run **show dbs** command and check our created database in **MongoDB Atlas** by **MongoDB compass**. Here we can also see our imported file through **MnogoDB compass**.

MongoDB

<https://www.mongodb.com/docs/manual/>

- In MongoDB Atlas, the term "driver" refers to the software component that allows a programming language or application to interact with a MongoDB database. MongoDB is a NoSQL database, and to interact with it programmatically, we need a specific driver that is compatible with the programming language you are using.

MongoDB provides official drivers for various programming languages, such as Python, Java, JavaScript (Node.js), C#, and many others. These drivers are essential for enabling communication between our application code and the MongoDB database.

When we work with MongoDB Atlas, which is a fully managed cloud database service for MongoDB, we typically use the MongoDB driver for our chosen programming language to connect to and interact with our MongoDB Atlas cluster.

- **The explain() method:** in MongoDB is used to provide information about the query execution plan, index usage, and other details related to the performance of a specific query. This method helps developers and database administrators analyze and optimize queries for better performance.

Syntax: `db.collection.find(query).explain("executionStats");`

Example: `db.students.find({ "score": { $gt: 80 } }).explain("executionStats");`

- **What is MQL (MongoDB Query Language)?**

MongoDB Query Language (MQL) is MongoDB's proprietary language designed for data retrieval and manipulation within the MongoDB database. It allows users to specify conditions using operators such as \$eq (equals), \$ne (not equals), \$gt (greater than), \$lt (less than), and others. MQL supports multiple bitwise operators such as \$and, \$or, and \$not, enabling the creation of complex queries.

MongoDB Query Language (MQL) Overview:

- **Purpose:** Designed for data retrieval and manipulation in MongoDB.
 - **Functionality:** Enables users to specify conditions, execute aggregations, update documents, and perform deletions.
 - **Syntax:** Simple and intuitive, resembling JSON-like structures.
 - **Operators:** Includes various operators such as \$eq (equals), \$ne (not equals), \$gt (greater than), \$lt (less than), and more.
 - **Bitwise Operators:** Supports bitwise operators like \$and, \$or, and \$not for creating complex queries.
 - **Aggregation:** Empowers users with powerful aggregation capabilities, often utilized in aggregation pipelines.
- **Explain the basic syntax of a MongoDB query.**

The basic syntax of a MongoDB query is expressed using the find() method, which is commonly used for retrieving documents from a MongoDB collection.

The general syntax looks like:

`db.collectionName.find(query, projection)`

- **db:** Refers to the current database.
 - **collectionName:** Specifies the name of the collection from which data is to be retrieved.
 - **find():** Method used for querying documents in a collection. The **find()** method takes two optional parameters.
 - **query (optional):** Defines the conditions that the documents must meet to be included in the result. It is a JSON-like object specifying the criteria.
 - **projection (optional):** Specifies which fields to include or exclude from the result. It is also a JSON-like object.
- Example:** `db.collectionName.find({ field: value }, { fieldName: 1, _id: 0 })`

- **Differentiate between the find() and findOne() methods.**

find() Method:

Returns a cursor.

Retrieves multiple documents and suitable when expecting multiple results.

Example: `db.collectionName.find({ field: value })`

findOne() Method:

Returns a single document as an object.

MongoDB

<https://www.mongodb.com/docs/manual/>

Retrieves only one document and suitable when expecting or needing only one result.

Example: `db.collectionName.findOne({ field: value })`

- **How do you query for documents with a specific field in MongoDB?**

In MongoDB, we can query for documents with a specific field using the `find()` method and specifying the field and its value in the query criteria.

Example: `db.collectionName.find({ fieldName: value })`

- **Explain the concept of embedding documents in MongoDB.**

Embedding documents in MongoDB involves storing one document within another, either as a field or array. This allows for hierarchical data representation without the need for complex joins. Use cases include one-to-one or one-to-many relationships, offering improved read performance. However, consider document size limits and update anomalies in our design.

- **What is sharding in MongoDB?**

Sharding in MongoDB is a method for horizontally scaling a database by distributing data across multiple servers or clusters. It involves breaking data into smaller chunks (shards) and distributing them based on a chosen shard key. This enables MongoDB to handle large datasets and high workloads by leveraging multiple machines in parallel. Sharding is beneficial for applications with substantial data volumes and performance requirements.

- **What are the limitations of MongoDB?**

MongoDB is a popular NoSQL database with many advantages, but it also has some limitations.

- **Transaction Support:** MongoDB has limited support for multi-document transactions.
- **Memory Consumption:** MongoDB can have high memory usage, and it's recommended that the working set fit in RAM.
- **Complex Joins:** MongoDB lacks support for traditional SQL-style joins.
- **Indexing Overhead:** Maintaining indexes in MongoDB can impact write performance.
- **Schema Flexibility:** While MongoDB's schema-less structure provides flexibility, it can also lead to challenges in enforcing data consistency.
- **Storage Space:** MongoDB may use more storage space compared to some other databases due to its document-oriented nature.
- **Aggregation Framework:** There can be a learning curve associated with MongoDB's Aggregation Framework.
- **Atomic Operations:** Some operations in MongoDB are not atomic across multiple documents.
- **Global Lock (pre-4.0):** Earlier versions of MongoDB had a global write lock, but this has been improved in recent releases.

Remember, these limitations do not necessarily make MongoDB a poor choice. The right database depends on the specific requirements of the application it's intended to support. MongoDB is still widely used due to its high performance, availability, and scalability.

- **How do you model relationships between entities in MongoDB?**

In MongoDB, relationships between entities can be modeled using either Embedded or Referenced relationships:

1. **Embedded Relationships:**
 - Store related data within a single document.
 - Suitable for one-to-one or one-to-few relationships.
 - Provides better performance for read operations but may lead to data duplication.
2. **Referenced Relationships:**
 - Store references (usually `ObjectId`) to related documents.
 - Suitable for one-to-many or many-to-many relationships.
 - Reduces data duplication but may require additional queries to retrieve related data.

This is a fundamental aspect of MongoDB's document-oriented nature, allowing it to efficiently model complex hierarchical relationships with a mix of structured and unstructured data. It's also a key factor in MongoDB's flexibility and performance characteristics.

MongoDB

<https://www.mongodb.com/docs/manual/>

- **What is denormalization, and when is it appropriate to use in MongoDB?**

Denormalization in MongoDB involves duplicating data across multiple documents or collections to optimize query performance. It is appropriate to use denormalization when:

1. **Read Performance is Critical:** When there is a need for fast read operations, denormalization can be employed to reduce the number of queries and improve response times.
2. **Query Patterns are Known:** If the application has well-defined and predictable query patterns, denormalization can be tailored to suit those specific queries, minimizing the need for complex joins or multiple database calls.
3. **Data Consistency can be Managed:** If the application can handle occasional inconsistencies in data due to denormalization (e.g., through background jobs to update duplicated data), it may be a suitable choice.

Denormalization should be used judiciously, considering trade-offs such as increased storage requirements and the need for careful maintenance of data consistency.

- **What is sharded cluster in mongo db? How does MongoDB distribute data in a sharded cluster? Also explain the role of a shard key in sharding.**

1. **Sharded Cluster in MongoDB:** A sharded cluster is a distributed database architecture in MongoDB designed to horizontally scale data across multiple servers (shards). It allows MongoDB to handle large amounts of data and high write and read loads by distributing the data across multiple machines.
2. **Data Distribution in Sharded Cluster:** MongoDB distributes data in a sharded cluster by dividing the dataset into chunks and distributing these chunks across different shards. Each shard is responsible for a subset of the data, and MongoDB's balancer ensures an even distribution of data among the shards.
3. **Role of Shard Key:** A shard key is a field that determines the distribution of data across shards. MongoDB uses the shard key to divide the dataset into chunks and distribute these chunks across shards. A well-chosen shard key is crucial for achieving an even distribution of data and optimal performance in a sharded cluster. It influences how data is distributed, so selecting an appropriate shard key is essential for efficient sharding.

- **What is role-based access control (RBAC) in MongoDB?**

Role-Based Access Control (RBAC) in MongoDB is a security mechanism that restricts database access by assigning specific roles to users. Each role defines a set of privileges, such as read or write permissions, and users are granted roles based on their responsibilities. RBAC helps enforce the principle of least privilege, ensuring users only have the necessary permissions for their tasks. This is a key aspect of MongoDB's security model, allowing fine-grained control over who can access what data and what operations they can perform.

- **How do you secure a MongoDB database?**

Authentication: Use strong usernames and passwords.

Authorization: Implement role-based access control (RBAC).

Network Security: Bind to specific IP addresses and use firewalls to control access.

Encryption: Enable SSL/TLS for data in transit and consider encryption for data at rest.

Audit Logging: Enable audit logs to track activities.

Updates and Patching: Keep MongoDB updated.

Disable Unused Features: Turn off unnecessary services.

Secure Deployment: Follow deployment best practices.

Backup and Recovery: Regularly backup data and plan for recovery.

Monitoring: Use monitoring tools to detect anomalies.

These are all important aspects of securing a MongoDB database.

- **Explain the concept of authentication in MongoDB.**

Authentication in MongoDB involves validating the identity of users attempting to access the database. Users must provide a valid combination of a username and password to authenticate. MongoDB supports various authentication mechanisms, including SCRAM (Salted Challenge Response Authentication Mechanism) and LDAP (Lightweight Directory Access Protocol), to secure database access.

This is a crucial aspect of MongoDB's security model, ensuring that only authorized users can access the database.

MongoDB

<https://www.mongodb.com/docs/manual/>

In conclusion, MongoDB stands out as a versatile NoSQL database, offering flexibility in data modeling, scalability through sharding, and robust features for handling diverse data types. Its document-oriented structure, support for horizontal scaling, and dynamic schema make it well-suited for modern, data-intensive applications. While embracing MongoDB's strengths, it is crucial to implement robust security measures, adhere to best practices, and stay informed about updates to leverage its full potential in building scalable and efficient systems.



ॐ ऐं ह्रीं क्लीं लक्ष्मी नारायणाय नमः
