**1. Introduction to MongoDB**

* What is MongoDB?
* Features of MongoDB
* Types of Databases (SQL vs NoSQL)
* Why does MongoDB use BSON?
* BSON Advantages
* Alternatives to MongoDB (Cassandra, Redis, DynamoDB, HBase, OrientDB)

**2. Data Types in MongoDB**

* Data Types in MongoDB
* BSON Data Types
* ObjectId Structure and Size
* Embedded Documents
* Reference documents

**3. Basic MongoDB Operations**

* Collections
* Insert vs Save
* Update vs UpdateOne vs UpdateMany
* Delete Operations (DeleteOne, DeleteMany)
* Basic Query Operations (find, findOne)
* Cursors
* Admin Database
* How to List Collections
* How to Modify a Collection Name(db.collection.renameCollection())

**4. Indexing in MongoDB**

* What is Indexing?
* Single Field Index
* Compound Index
* Multi-Key Index
* Geospatial Index
* Text Index
* Hashed index
* Covered Queries
* How to Create Indexes (db.collection.createIndex())
* Indexing Best Practices
* Clustered Index vs Non-Clustered Index
* Clustered Collections

**5. Aggregation Framework**

* Aggregation Framework Overview
* Basic Aggregation Pipeline Stages:
  + $match
  + $group
  + $sort
  + $project
  + $limit
* Complex Aggregation Operators:
  + $or, $in, $exists
  + $facet
  + $lookup
  + $merge
  + $unwind
  + $addToSet, $push, $pull, $pop
  + $all, $nin, $ne
  + $cond, $expr
* Aggregation Workouts (Exercises)
* Map-Reduce vs Aggregation Framework
* Covered Query
* single purpose aggr
* max size of mongo
* $rename

**6. Data Modeling**

* Relational vs Embedded Data Modeling
* Normalization vs Denormalization
* When to Use Embedded Documents
* Schema Design Best Practices

**7. CRUD Operations in MongoDB**

* Create, Read, Update, Delete (CRUD) Overview
* BulkWrite Operations
* Upsert Operation
* save() vs insert() Operations
* Aggregation in CRUD

**8. Advanced Querying in MongoDB**

* Advanced Query Syntax:
  + $regex (using regex in MongoDB)
  + $expr, $elemMatch
  + $exists

**9. Replication**

* What is Replication?
* Primary and Secondary Replica Set
* How Many Nodes in a Replica Set?
* Voting in Replication
* Difference Between GridFS and Sharding

**10. Sharding**

* What is Sharding?
* Components of Sharding
* Query Routing in Sharding
* Advantages and Disadvantages of Sharding
* Sharding vs Replication
* Sharding Best Practices
* CAP Theorem
* Capped Collections
* How to Create a Capped Collection
* Sharding Disadvantages

**11. GridFS**

* What is GridFS?
* Difference Between GridFS and Sharding
* GridFS vs Traditional File Storage

**12. Transactions and Batch Operations**

* Transactions in MongoDB
* ACID Compliance
* Batch Sizing
* Upsert Operations
* Use Cases for Transactions

**13. Backup and Restore**

* MongoDB Backup and Restore Commands
* Backup Best Practices
* Restore Best Practices
* Backup storages

**Session 14 – 17 not important it’s only use in 6th Week.**

**14. Security Best Practices**

* Authentication and Authorization in MongoDB
* Role-Based Access Control (RBAC)
* Encryption Best Practices
* Secure Network Configuration
* Auditing and Logging in MongoDB

**15. Performance Tuning**

* Indexing for Performance
* Query Optimization
* Caching Strategies
* Load Balancing
* Performance Tuning Best Practices

**16. MongoDB Atlas**

* Overview of MongoDB Atlas
* Setting Up MongoDB Atlas
* Atlas Clustering
* Security Features in Atlas

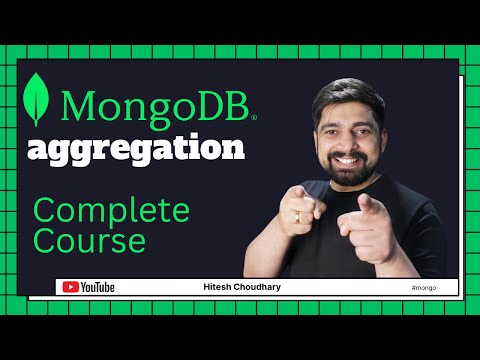
**17. Document Validation**

* Schema Validation in MongoDB
* Custom Validation Rules
* Validation Best Practices

**18. Miscellaneous Topics**

* CAP Theorem
* TTL (Time to Live)
* Data Redundancy
* Clustered Collections
* Materialized Views
* View collections
* Decrement Operations
* Alternatives to MongoDB
  + Cassandra
  + Redis
  + DynamoDB
  + HBase
  + OrientDB

Aggregation Preferred video:-

[](https://www.youtube.com/watch?v=vx1C8EyTa7Y)

CRUD BASIC VIDEO:-

[](https://www.youtube.com/watch?v=_nQ4FCd8_Sg)

**1. Introduction to MongoDB**

* What is MongoDB?

**MongoDB** is a modern, open-source NoSQL database that handles lots of unstructured data. Instead of using tables like traditional databases, it stores data in flexible, JSON-like documents called BSON. This means you can easily change the structure of your data without any issues. MongoDB is great for applications that need to process large amounts of data quickly, like real-time analytics and big data projects. It’s also easy to use and works well with modern development tools, making it a popular choice for developers.

* Features of MongoDB
  + **Flexible Schema**: No need for a fixed structure; documents can vary.
  + **Document Storage**: Stores data in BSON format, supporting nested documents and arrays.
  + **High Performance**: Fast read and write operations with indexing and in-memory processing.
  + **Scalability**: Can handle large datasets by distributing data across multiple servers (sharding) and ensuring data redundancy (replication).
  + **Powerful Queries**: Supports complex queries, filtering, sorting, and text search.
  + **Aggregation**: Allows for advanced data processing and transformations.
  + **Indexing**: Various types of indexes to speed up queries.
  + **Data Integrity**: Ensures data accuracy with atomic operations and supports transactions for complex operations.
  + **High Availability**: Keeps data accessible even if some servers fail, using replica sets.
  + **Developer-Friendly**: Easy to use with a straightforward API and support for many programming languages.
* Types of Databases (SQL vs NoSQL)

### SQL Databases

* **Structured Data**: SQL databases store data in tables with rows and columns, similar to a spreadsheet.
* **Fixed Schema**: You need to define the structure of your data (schema) before you can store it.
* **Relational**: Data is organized in a way that allows relationships between different tables.
* **ACID Compliance**: Ensures reliable transactions with properties like Atomicity, Consistency, Isolation, and Durability.
* **Vertical Scalability**: Typically scaled by increasing the power of a single server (e.g., adding more CPU, RAM).

**Examples**: MySQL, PostgreSQL, Oracle, SQL Server.

### NoSQL Databases

* **Flexible Data**: NoSQL databases store data in various formats like documents, key-value pairs, graphs, or wide-columns.
* **Schema-less**: You don’t need to define the structure of your data in advance, allowing for more flexibility.
* **Non-Relational**: Data is often stored without strict relationships, making it easier to handle unstructured data.
* **High Scalability**: Designed to scale out horizontally by adding more servers.
* **Eventual Consistency**: Some NoSQL databases prioritize availability and partition tolerance over immediate consistency.

**Examples**: MongoDB, Cassandra, Redis, Neo4j.

### Key Differences

* **Structure**: SQL uses structured tables, while NoSQL uses flexible formats.
* **Schema**: SQL requires a predefined schema; NoSQL does not.
* **Scalability**: SQL scales vertically; NoSQL scales horizontally.
* **Use Cases**: SQL is great for complex queries and transactions; NoSQL is ideal for large volumes of unstructured data and real-time applications.
* Why does MongoDB use BSON?

“MongoDB uses BSON (Binary JSON) because it is a binary format that is more efficient for storage and retrieval, supports a wider range of data types, and allows for faster parsing and flexibility in representing complex data structures.”

1. BSON is a binary format, which means it can store data more compactly than plain text JSON. This helps in saving storage space.
2. BSON is designed to be fast to encode and decode. This makes data retrieval and storage operations quicker.
3. BSON supports more data types than JSON, such as dates and binary data.This allows MongoDB to handle a wider variety of data efficiently.
4. BSON is designed to be traversable, meaning MongoDB can easily navigate through the data to perform operations like queries and indexing.
5. BSON maintains the order of keys in documents, which can be important for certain applications

* BSON Advantages

### Advantages of BSON

1. **Efficiency**: BSON is a binary format, which makes it faster to read and write compared to text-based formats like JSON
2. **Compactness**: It generally results in smaller file sizes, saving storage space and improving transmission speeds
3. **Rich Data Types**: BSON supports a wider range of data types, including dates and binary data, which JSON does not
4. **Speed**: The binary encoding allows for quicker parsing and efficient data traversal
5. **Flexibility**: It supports nested documents and arrays, making it easier to represent complex data structures

### Disadvantages of BSON

1. **Space Efficiency**: While BSON is compact, it can sometimes be less space-efficient than JSON due to additional metadata.
2. **Human Readability**: BSON is not human-readable, which can make debugging and manual data inspection more challenging.
3. **Complexity**: The binary format can be more complex to work with compared to the simpler, text-based JSON.

* Alternatives to MongoDB (Cassandra, Redis, DynamoDB, HBase, OrientDB)

**2. Data Types in MongoDB**

* Data Types in MongoDB

MongoDB supports a variety of data types to handle different kinds of information. Here are some of the key data types:

1. **String**: Used to store text data. Must be UTF-8 valid.
2. **Integer**: Stores numerical values. Can be 32-bit or 64-bit.
3. **Boolean**: Stores true or false values.
4. **Double**: Used for floating-point numbers.
5. **Date**: Stores the current date and time in a format compatible with the ISODate.
6. **Array**: Stores a list of values, which can be of different data types.
7. **Object**: Used to store embedded documents.
8. **Null**: Represents a null value.
9. **ObjectId**: A unique identifier for documents.
10. **Binary Data**: Stores binary data.
11. **Regular Expression**: Used for pattern matching.
12. **Timestamp**: Stores a timestamp.
13. **Decimal128**: Stores high-precision decimal values.
14. **Min/Max Keys**: Used to compare a value against the lowest and highest BSON elements.

These data types allow MongoDB to handle a wide range of data and provide flexibility in how you store and manage your information.

* BSON Data Types
  + **Double**: 64-bit floating point number.
  + **String**: UTF-8 encoded string.
  + **Object**: Embedded document.
  + **Array**: Array of values.
  + **Binary Data**: Binary data.
  + **Undefined**: Deprecated.
  + **ObjectId**: Unique identifier for documents.
  + **Boolean**: true or false value.
  + **Date**: Date and time.
  + **Null**: Null value.
  + **Regular Expression**: Regular expression for pattern matching.
  + **DBPointer**: Deprecated.
  + **JavaScript**: JavaScript code.
  + **Symbol**: Deprecated.
  + **JavaScript with Scope**: JavaScript code with a scope (deprecated in MongoDB 4.4).
  + **32-bit Integer**: 32-bit integer.
  + **Timestamp**: Special internal timestamp.
  + **64-bit Integer**: 64-bit integer.
  + **Decimal128**: High-precision decimal value.
  + **Min Key**: Minimum key value.
  + [**Max Key**: Maximum key value](https://www.mongodb.com/docs/manual/reference/bson-types/)
* ObjectId Structure and Size

An **ObjectId** in MongoDB is a unique identifier for documents. It is 12 bytes in size and consists of the following components:

* **4-byte Timestamp**: Represents the creation time of the ObjectId, measured in seconds since the Unix epoch.
* **5-byte Random Value**: Generated once per process, unique to the machine and process.
* [**3-byte Counter**: An incrementing counter, initialized to a random value](https://www.mongodb.com/docs/manual/reference/method/ObjectId/)
* Embedded Documents

Embedded documents in MongoDB are documents stored within other documents, creating a nested structure. This approach is useful for storing related data together, making it easier to access and manage.

### Example:

Imagine you have a user document that includes the user’s address. Instead of storing the address in a separate collection, you can embed it directly within the user document:

**JSON**

{

"\_id": 111111,

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

"name": {

"given": "Jane",

"family": "Han"

},

"address": {

"street": "111 Elm Street",

"city": "Springfield",

"state": "Ohio",

"country": "US",

"zip": "00000"

}

}

### Benefits:

* **Efficiency**: Accessing related data is faster because it’s stored together.
* **Simplicity**: Easier to manage and update related data within a single document.
* **Consistency**: Ensures that related data is always kept together, reducing the risk of inconsistencies.

### When to Use:

* When related data is frequently accessed together.
* When the data structure is relatively simple and doesn’t require complex relationships.

### When Not to Use:

* When the embedded data grows too large, making the document unwieldy.
* When the data has complex relationships that are better managed with references.

**3. Basic MongoDB Operations**

* Collections

A **collection** in MongoDB is like a table in a relational database. It’s a group of documents. Unlike tables, collections in MongoDB are schema-less, meaning documents within a collection can have different fields.

* Insert vs Save
  + **Insert**: Adds a new document to a collection. If the document already exists, it will not be added again.
  + **Save**: If the document has an \_id field and it matches an existing document, save will update that document. If there’s no match, it will insert the document as a new one.
* Update vs UpdateOne vs UpdateMany
* **Update**: Modifies existing documents in a collection. By default, it updates the first document that matches the criteria.
* **UpdateOne**: Updates a single document that matches the criteria.
* **UpdateMany**: Updates all documents that match the criteria.
* Delete Operations (DeleteOne, DeleteMany)
  + **DeleteOne**: Removes a single document that matches the criteria.
  + **DeleteMany**: Removes all documents that match the criteria.
* Basic Query Operations (find, findOne)
  + **find**: Retrieves multiple documents that match the query criteria.
  + **findOne**: Retrieves a single document that matches the query criteria.

**Comparison Operators**

1. **$eq**: Matches values that are equal to a specified value.
2. **$ne**: Matches values that are not equal to a specified value.
3. **$gt**: Matches values that are greater than a specified value.
4. **$gte**: Matches values that are greater than or equal to a specified value.
5. **$lt**: Matches values that are less than a specified value.
6. **$lte**: Matches values that are less than or equal to a specified value.
7. **$in**: Matches any of the values specified in an array.
8. **$nin**: Matches none of the values specified in an array.

**Logical Operators**

1. **$and**: Joins query clauses with a logical AND, returning all documents that match the conditions of both clauses.
2. **$or**: Joins query clauses with a logical OR, returning all documents that match the conditions of either clause.
3. **$not**: Inverts the effect of a query expression and returns documents that do not match the query expression.
4. **$nor**: Joins query clauses with a logical NOR, returning all documents that fail to match both clauses.

**Element Operators**

1. **$exists**: Matches documents that have the specified field.
2. **$type**: Matches documents that have a field of the specified type.

**Evaluation Operators**

1. **$regex**: Matches documents where the value of a field matches a specified regular expression.
2. **$expr**: Allows the use of aggregation expressions within the query language.
3. **$jsonSchema**: Validates documents against the given JSON Schema.

**Array Operators**

1. **$all**: Matches arrays that contain all elements specified in the query.
2. **$elemMatch**: Matches documents that contain an array field with at least one element that matches all the specified query criteria.
3. **$size**: Matches any array with the specified number of elements.

**Geospatial Operators**

1. **$geoWithin**: Selects documents with geospatial data that exist entirely within a specified shape.
2. **$geoIntersects**: Selects documents with geospatial data that intersect with a specified shape.
3. **$near**: Returns documents in order of proximity to a specified point.

* Cursors

A **cursor** is an object that allows you to iterate over the results of a query. When you use find, it returns a cursor, which you can use to access each document one by one.

* Admin Database

The **admin database** is a special database in MongoDB that holds administrative information and commands. It’s used for tasks like managing users and roles, and performing server-side operations.

* How to List Collections

To list all collections in a database, you can use the listCollections command or the show collections command in the MongoDB shell.

* How to Modify a Collection Name
* To rename a collection, you can use the renameCollection command. For example:

db.oldCollectionName.renameCollection("newCollectionName")

**4. Indexing in MongoDB**

* What is Indexing?
* **Indexes** are special data structures that store a small portion of the collection’s data in an easy-to-traverse form. They are similar to the index in a book, which helps you quickly find the information you need without having to read through the entire book.

### Key Points about Indexes:

* **Purpose**: They make it faster to retrieve documents from a collection by reducing the amount of data MongoDB needs to scan.
* **Types**: MongoDB supports various types of indexes, including single field, compound, multi-key, text, and geospatial indexes.
* **Creation**: You can create an index on a collection using the createIndex method.
* **Usage**: When you query a collection, MongoDB uses the index to quickly locate the required documents.

For example, if you have a collection of books and you frequently search by the author’s name, you can create an index on the author field to speed up these queries.

### Default \_id Index:

every collection automatically has a default index on the \_id field. This index is created when the collection is created and ensures that each document in the collection has a unique identifier.

* Single Field Index
  + A **single field index** is an index on a single field of a document. For example, if you frequently query by the name field, you can create an index on name to speed up those queries:

db.collection.createIndex({ name: 1 })

* Compound Index
  + A **compound index** is an index on multiple fields. This is useful for queries that filter on multiple fields. For example:

db.collection.createIndex({ name: 1, age: -1 })

* Multi-Key Index
  + A **multi-key index** is used for indexing fields that hold arrays. MongoDB creates an index entry for each element of the array. For example:

db.collection.createIndex({ tags: 1 })

* Geospatial Index
  + A **geospatial index** is used for querying geospatial data. MongoDB supports 2D and 2DSphere indexes for different types of geospatial queries. For example:

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

* Text Index
  + A **text index** is used for text search queries. It indexes the content of string fields for efficient text search. For example:

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

* Covered Queries

A **covered query** is a query where all the fields in the query are part of an index. This means MongoDB can satisfy the query using only the index, without scanning any documents. This can significantly improve performance.

* How to Create Indexes (db.collection.createIndex())

To create an index, use the createIndex method:

db.collection.createIndex({ field: 1 })

* Indexing Best Practices
  + **Analyze Query Patterns**: Create indexes based on the fields that are frequently queried.
  + **Limit the Number of Indexes**: Each index consumes disk space and affects write performance.
  + **Use Compound Indexes Wisely**: Ensure the order of fields in compound indexes matches the query patterns.
  + **Monitor Index Usage**: Use tools like MongoDB Atlas Performance Advisor to monitor and optimize index usage.
* Clustered Index vs Non-Clustered Index
  + **Clustered Index**: MongoDB does not support clustered indexes in the traditional sense. However, the \_id field in MongoDB is automatically indexed and can be considered similar to a clustered index.
  + **Non-Clustered Index**: All other indexes in MongoDB are non-clustered. They store a reference to the actual data rather than the data itself.

**5. Framework**

* Aggregation Framework Overview

The Aggregation Framework in MongoDB is a powerful tool for performing data aggregation operations, such as filtering, grouping, sorting, transforming, and calculating data across multiple documents in a collection. It allows you to process data and derive insights without needing to transfer data to an external application for computation.

**Key Concepts of Aggregation Framework:**

1. **Aggregation Pipeline**:
   * The aggregation framework works like a pipeline where data passes through various stages, with each stage performing an operation on the data.
   * The output of one stage becomes the input for the next stage.

* Basic Aggregation Pipeline Stages:

**1. $match:**

* The $match stage filters documents in a collection based on specified criteria, similar to the find() query. It reduces the number of documents that enter the next stage of the pipeline.
* **Example**: Find users from the USA.

db.user.aggregate([

{

$match: { "country": "USA" } // Filters documents where 'country' is 'USA'

}

])

**2. $group:**

* The $group stage groups documents by a specified field or fields and performs aggregation operations such as $sum, $avg, $max, etc., on each group.
* **Example**: Group users by favoriteFruit and calculate the total count of users in each group.

db.user.aggregate([

{

$group: {

\_id: "$favoriteFruit", // Group by 'favoriteFruit'

count: { $sum: 1 } // Count the number of users in each group

}

}

])

**3. $sort:**

* The $sort stage sorts documents by a specified field or fields, either in ascending (1) or descending (-1) order.
* **Example**: Sort users by age in descending order.

db.user.aggregate([

{

$sort: { "age": -1 } // Sort documents by 'age' in descending order

}

])

**4. $project:**

* The $project stage reshapes each document by including, excluding, or transforming fields. It can be used to create new fields or modify existing ones.
* **Example**: Include only the name and age fields in the output.

db.user.aggregate([

{

$project: {

name: 1, // Include the 'name' field

age: 1 // Include the 'age' field

}

}

])

* You can also perform transformations within $project:

db.user.aggregate([

{

$project: {

name: 1, // Include 'name'

ageInFiveYears: { $add: ["$age", 5] } // Add 5 to the 'age' field and store it as 'ageInFiveYears'

}

}

])

**5. $limit:**

* The $limit stage restricts the number of documents passed to the next stage of the pipeline. It is useful when you need to return a specific number of documents, such as in pagination.
* **Example**: Limit the result to the first 5 documents.

db.user.aggregate([

{

$limit: 5 // Return only the first 5 documents

}

])

* Complex Aggregation Operators:

**1. $or, $in, $exists**

* **$or**: Matches documents where at least one of the conditions in the array is true.
  + **Example**: Find users who are either from the USA or are older than 30.

db.user.find({

$or: [

{ country: "USA" },

{ age: { $gt: 30 } }

]

})

* **$in**: Matches any documents where the field’s value is in the specified array.
  + **Example**: Find users whose favorite fruit is either "Apple" or "Banana."

db.user.find({

favoriteFruit: { $in: ["Apple", "Banana"] }

})

* **$exists**: Checks if a field exists in a document.
  + **Example**: Find users that have an email field.

db.user.find({

email: { $exists: true }

})

**2. $facet**

* **$facet**: Allows running multiple aggregation pipelines within a single query and outputs a documentcur containing the results of all pipelines.
  + **Example**: Run two facets: one to group by favoriteFruit and count users, and another to get the average age of users.

db.user.aggregate([

{

$facet: {

fruitCounts: [

{ $group: { \_id: "$favoriteFruit", count: { $sum: 1 } } }

],

averageAge: [

{ $group: { \_id: null, avgAge: { $avg: "$age" } } }

]

}

}

])

**3. $lookup**

* **$lookup**: Performs a left outer join between two collections. Useful for joining data from different collections in MongoDB.
  + **Example**: Join orders collection with users collection to include user details in each order.

db.orders.aggregate([

{

$lookup: {

from: "users", // The collection to join

localField: "userId", // Field from 'orders'

foreignField: "\_id", // Field from 'users'

as: "userDetails" // Name for the resulting joined field

}

}

])

**4. $merge**

* **$merge**: Writes the results of an aggregation pipeline into a specified collection, replacing, merging, or inserting new documents.
  + **Example**: Merge the aggregation results into a collection called aggregatedResults.

db.orders.aggregate([

// ... Your aggregation pipeline ...

{ $merge: "aggregatedResults" } // Merge the output into 'aggregatedResults'

])

**5. $unwind**

* **$unwind**: Deconstructs an array field from the input documents to output a document for each element in the array.
  + **Example**: Unwind the hobbies array to have one document per hobby.

db.user.aggregate([

{ $unwind: "$hobbies" }

])

**6. $addToSet, $push, $pull, $pop**

* **$addToSet**: Adds a value to an array, only if the value does not already exist in the array (like a set).
  + **Example**: Add a hobby to a user’s hobbies array, only if it doesn’t already exist.

db.user.updateOne(

{ \_id: userId },

{ $addToSet: { hobbies: "reading" } })

* **$push**: Adds a value to an array, regardless of whether it already exists.
  + **Example**: Push a new hobby to a user’s hobbies array.

db.user.updateOne(

{ \_id: userId },

{ $push: { hobbies: "gaming" } }

)

* **$pull**: Removes all instances of a value from an array.
  + **Example**: Remove a specific hobby from the hobbies array.

db.user.updateOne(

{ \_id: userId },

{ $pull: { hobbies: "gaming" } }

)

* **$pop**: Removes the first or last element from an array.
  + **Example**: Pop the last hobby from the array.

db.user.updateOne(

{ \_id: userId },

{ $pop: { hobbies: 1 } } // Use -1 for the first element

)

**7. $all, $nin, $ne**

* **$all**: Matches documents where the array field contains all the specified elements.
  + **Example**: Find users whose hobbies include both "reading" and "traveling."

db.user.find({

hobbies: { $all: ["reading", "traveling"] }

})

* **$nin**: Matches documents where the field’s value is not in the specified array.
  + **Example**: Find users whose favorite fruit is neither "Apple" nor "Banana."

db.user.find({

favoriteFruit: { $nin: ["Apple", "Banana"] }

})

* **$ne**: Matches documents where the field’s value is not equal to the specified value.
  + **Example**: Find users who do not live in the USA.

db.user.find({

country: { $ne: "USA" }

})

**8. $cond, $expr**

* **$cond**: A conditional operator that allows you to evaluate an expression and return a value based on the result (similar to an if-else statement).
  + **Example**: Use $cond to assign a field called status based on age.

db.user.aggregate([

{

$project: {

name: 1,

status: {

$cond: { if: { $gte: ["$age", 18] }, then: "Adult", else: "Minor" }

}

}

}

])

* **$expr**: Allows you to use aggregation expressions in a query directly.
  + **Example**: Find users where the age field is greater than the yearsOfExperience field.

db.user.find({

$expr: { $gt: ["$age", "$yearsOfExperience"] }

})

* Aggregation Workouts (Exercises)
* Map-Reduce vs Aggregation Framework

**Map-Reduce**

How it works: Map-Reduce involves two functions: map and reduce. The map function processes each document and emits key-value pairs. The reduce function then processes these pairs to aggregate the results.

Flexibility: It allows for complex operations using JavaScript, making it highly flexible.

Performance: Generally slower and less efficient compared to the Aggregation Framework, especially for large datasets.

Use Cases: Suitable for complex data processing tasks that require custom JavaScript functions.

db.sales.mapReduce(

function() { emit(this.category, this.amount); },

function(key, values) { return Array.sum(values); },

{ out: "total\_sales\_by\_category" }

);

**Aggregation Framework**

How it works: Uses a pipeline of stages to process data. Each stage transforms the documents as they pass through the pipeline.

Built-in Operators: Includes a variety of built-in operators for filtering, grouping, sorting, and transforming data.

Performance: More efficient and faster than Map-Reduce, especially for large datasets.

Use Cases: Ideal for most aggregation tasks due to its performance and ease of use.

db.sales.aggregate([

{ $group: { \_id: "$category", totalSales: { $sum: "$amount" } } }

]);

**When to Use Map-Reduce vs. Aggregation Framework**

* **Use Map-Reduce** when:
  + You need highly customized aggregation logic that cannot be easily achieved with the built-in operators of the Aggregation Framework.
  + You are dealing with extremely large datasets that can benefit from distributed processing across multiple nodes.
* **Use the Aggregation Framework** when:
  + You are performing standard data processing tasks such as filtering, grouping, sorting, and projections.
  + You want better performance and real-time processing.
  + You prefer working with MongoDB’s built-in aggregation operators for ease of use and simplicity.

**Conclusion**

Both Map-Reduce and the Aggregation Framework are powerful tools for data aggregation in MongoDB. The choice between them depends on your specific use case. For most standard data processing tasks, the Aggregation Framework is the better option due to its performance and ease of use. However, for more complex or highly customized data transformations, Map-Reduce may still be the appropriate choice.

* Covered Query

A **Covered Query** in MongoDB is a type of query where MongoDB can get all the information it needs from the index itself, without having to look at the actual documents in the collection. This makes the query much faster because MongoDB doesn't need to read any extra data from the disk.

#### **How does it work?**

To have a covered query, three things need to happen:

1. **All the fields** used in the query must be part of the index.
2. **The query only asks for fields** that are in the index (no extra fields).
3. **The index is used** for filtering, sorting, and retrieving the results.

#### **Example:**

Let's say you have a collection called users, and each document looks like this:

{

"name": "Alice",

"age": 25,

"email": "alice@example.com"

}

Now, you create an index on the name and age fields:

db.users.createIndex({ name: 1, age: 1 });

If you run the following query:

db.users.find({ name: "Alice" }, { name: 1, age: 1, \_id: 0 });

This query:

* Filters by name.
* Projects (returns) only the name and age fields.

Since both name and age are part of the index, MongoDB can get the results directly from the index without reading the full document. This makes the query a **covered query**.

#### **Why is it good?**

* **Faster queries**: Since MongoDB doesn’t need to fetch the actual documents, it saves time.
* **Less data to process**: MongoDB only works with the index, so it's quicker and uses fewer resources.

**6. Data Modeling**

**Relational vs Embedded Data Modeling**

* **Relational**: Data is stored in separate tables, and relationships are defined using foreign keys. Think of it like a spreadsheet where each sheet is a table, and you link them using unique IDs.
* **Embedded**: Data is stored within a single document. It’s like having all related information in one place, like a nested list or a JSON object.

**Normalization vs Denormalization**

* **Normalization**: Splitting data into multiple tables to reduce redundancy. It’s like organizing your files into different folders to avoid duplicates.
* **Denormalization**: Combining related data into a single table to improve read performance. It’s like putting all your important documents in one folder for quick access.

**When to Use Embedded Documents**

* Use embedded documents when:
  + - The data is closely related and often accessed together.
    - You want to avoid multiple queries to fetch related data.
    - The size of the embedded document is manageable and won’t exceed MongoDB’s document size limit (16MB).

**Schema Design Best Practices**

1. **Understand Your Queries**: Design your schema based on how your application will query the data.
2. **Embed When Possible**: Embed related data to reduce the number of queries.
3. **Use References When Necessary**: Use references for data that is frequently updated or shared across multiple documents.
4. **Avoid Deep Nesting**: Keep nesting to a minimum to avoid performance issues.
5. **Consider Indexes**: Create indexes on fields that are frequently queried to improve performance.

**7. CRUD Operations in MongoDB**

**Create, Read, Update, Delete (CRUD) Overview**

**CRUD** stands for **Create, Read, Update, and Delete**—the four basic operations of persistent storage in a database. In MongoDB, these operations are performed on documents within collections.

* **Create**: Inserting new documents into a collection.
  + Example: db.collection.insertOne({ name: "Alice", age: 25 })
* **Read**: Querying documents from a collection.
  + Example: db.collection.find({ age: 25 })
* **Update**: Modifying existing documents in a collection.
  + Example: db.collection.updateOne({ name: "Alice" }, { $set: { age: 26 } })
* **Delete**: Removing documents from a collection.
  + Example: db.collection.deleteOne({ name: "Alice" })

**BulkWrite Operations**

**BulkWrite** operations allow you to perform multiple write operations (insert, update, delete) in a single request. This can improve performance when dealing with large numbers of documents.

Example of bulkWrite in MongoDB:

db.collection.bulkWrite([

{ insertOne: { document: { name: "Bob", age: 30 } } },

{ updateOne: { filter: { name: "Alice" }, update: { $set: { age: 26 } } } },

{ deleteOne: { filter: { name: "John" } } }

]);

**Upsert Operation**

An **Upsert** is a combination of **Update** and **Insert**. If a document matching the filter does not exist, MongoDB will insert a new document. If it exists, MongoDB will update it.

Example of upsert in MongoDB:

db.collection.updateOne(

{ name: "Charlie" },

{ $set: { age: 28 } },

{ upsert: true }

);

In this example, if a document with name: "Charlie" exists, it will be updated. If it doesn’t, a new document will be created.

**save() vs insert() Operations**

* **insert()**: Adds a new document to a collection. If the document already exists (based on \_id), it will throw an error.
  + Example: db.collection.insert({ \_id: 1, name: "Alice" })
* **save()**: If the document already exists (based on \_id), save() will update it. If the document doesn’t exist, save() will insert it.
  + Example: db.collection.save({ \_id: 1, name: "Alice", age: 25 })

In MongoDB, save() is a convenient way to perform both insert and update operations, but it has been deprecated in favor of using insertOne() and updateOne() for clarity.

**Aggregation in CRUD**

**Aggregation** operations allow you to process data and perform transformations. Aggregation can be considered as an advanced form of query, where documents are processed in stages.

Example of an aggregation pipeline:

db.collection.aggregate([

{ $match: { age: { $gt: 20 } } },

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

{ $sort: { totalUsers: -1 } }

]);

In this example:

1. **$match** filters documents where age is greater than 20.
2. **$group** groups the documents by age and counts the total users in each group.
3. **$sort** sorts the groups by totalUsers in descending order.

This is a powerful way to perform operations like filtering, grouping, and sorting in a single query.

**8. Advanced Querying in MongoDB**

* Advanced Query Syntax:

### 1. ****$regex**** (Using Regular Expressions in MongoDB)

The $regex operator allows you to search for strings that match a particular pattern, defined using a regular expression (regex). It’s particularly useful for partial string matches or more complex text searches.

#### **Example:**

Find all users whose names start with "A":

db.users.find({ name: { $regex: '^A' } });

* **^A**: Matches any string that starts with the letter "A".

You can also add options like case-insensitivity:

db.users.find({ name: { $regex: '^a', $options: 'i' } });

* **i**: Case-insensitive matching.

### 2. ****$expr**** (Using Expressions in MongoDB)

The $expr operator allows you to use aggregation expressions within the find query. It’s useful when you need to compare fields within a document or perform calculations.

#### **Example:**

Find all users whose age is greater than score:

db.users.find({ $expr: { $gt: ["$age", "$score"] } });

* **$gt**: Checks if age is greater than score within the same document.

You can use any aggregation expression with $expr, including $add, $subtract, $and, etc.

### 3. ****$elemMatch**** (Matching Elements in an Array)

The $elemMatch operator is used to match documents that contain an array field, where at least one element in the array matches the specified condition(s).

#### **Example:**

Find all users who have a score array with at least one score greater than 80:

db.users.find({ scores: { $elemMatch: { $gt: 80 } } });

In this example, MongoDB will return documents where the scores array has at least one element greater than 80.

You can also match multiple conditions on a single element:

db.users.find({ scores: { $elemMatch: { $gt: 80, $lt: 90 } } });

This will return documents where the scores array has at least one element that is greater than 80 but less than 90.

### 4. ****$exists**** (Checking for Field Existence)

The $exists operator checks whether a particular field exists in a document. It’s useful for finding documents that either have or lack a specific field.

#### **Example:**

Find all users who have an email field:

db.users.find({ email: { $exists: true } });

* **$exists: true**: The field email must exist in the document.

Find all users who do not have a phone field:

db.users.find({ phone: { $exists: false } });

* **$exists: false**: The field phone must be absent from the document.
* Advanced Query Scenarios:
  + Find names ending with a specific letter (e.g., "e")
  + Find the average salary of all employees
  + Decrease salary by 1000 if a developer doesn’t know HTML (skills array)
  + Name of the person with the maximum salary
* Practice Exercises

**9. Replication**

**1. What is Replication in MongoDB?**

**Replication** in MongoDB is the process of synchronizing data across multiple servers. It provides redundancy and high availability by maintaining multiple copies of the same data. If one server fails, another can take over without data loss, ensuring that your application remains available even during server failures or maintenance.

Replication is achieved through **Replica Sets**, which are groups of MongoDB servers that maintain the same data set. Replica Sets provide automatic failover, data redundancy, and recovery options.

**2. Primary and Secondary Replica Set**

* **Primary**: The **primary** node in a replica set is the main server that receives all write operations. It accepts updates, inserts, and deletes, and replicates these changes to the secondary nodes. Applications connect to the primary node for all write operations.
* **Secondary**: **Secondary** nodes replicate data from the primary node. They hold read-only copies of the data and can be used for read operations, improving query performance. Secondary nodes help distribute the read load and act as backups in case the primary node fails.

When the primary node fails, one of the secondary nodes is automatically elected as the new primary.

Replication in MongoDB is a process that ensures data is copied and maintained across multiple servers. This helps in achieving high availability and data redundancy, meaning your data is safe even if one server fails.

### Simple Explanation:

* **Replication**: The process of copying data from one MongoDB server (primary) to other servers (secondaries).
* **Replica Set**: A group of MongoDB servers that maintain the same data set. It includes one primary node and multiple secondary nodes.

### How It Works:

1. **Primary Node**: Handles all write operations.
2. **Secondary Nodes**: Replicate the data from the primary node.
3. **Failover**: If the primary node fails, one of the secondary nodes is automatically elected as the new primary.

### Example:

Let’s say you have a users collection in your MongoDB database. You set up a replica set with one primary and two secondary nodes.

1. **Primary Node**: Receives all write operations.
2. **Secondary Nodes**: Continuously replicate the data from the primary node.

If you insert a new user into the users collection:

**JavaScript**

db.users.insertOne({ name: "Rimshan", age: 25 })

AI-generated code. Review and use carefully. [More info on FAQ](https://www.bing.com/new#faq).

This data is immediately replicated to the secondary nodes. If the primary node goes down, one of the secondary nodes will become the new primary, ensuring that your application continues to function without interruption.

### Real-Life Example:

Imagine a popular e-commerce website. To ensure that the website remains available even during server failures, the company uses MongoDB replication. They set up a replica set with servers located in different geographical regions. This way, if one server fails due to a hardware issue or a natural disaster, another server in a different location can take over, ensuring that customers can still access the website and make purchases.

**3. How Many Nodes in a Replica Set?**

A typical **MongoDB replica set** consists of **three nodes**:

1. One **primary** node.
2. Two **secondary** nodes.

However, a replica set can have up to **50 nodes**, with a maximum of **7 voting members**. The number of nodes can vary based on the need for redundancy, availability, and load balancing. For most production environments, a 3-node replica set is the most common setup.

**4. Voting in Replication**

In MongoDB replication, **voting** is part of the replica set's **election process**. When the primary node becomes unavailable, an election is held to choose a new primary. Only the voting members of the replica set participate in the election process.

* A replica set can have up to **7 voting members**.
* Voting members include both primary and secondary nodes.
* The node that gets a majority (more than half) of the votes is elected as the new primary.

Voting helps MongoDB ensure that there's a consistent primary node and that the replica set remains operational.

**5. Difference Between GridFS and Sharding**

Both **GridFS** and **Sharding** are MongoDB features used for handling large data, but they serve different purposes:

* **GridFS**: GridFS is a specification for storing and retrieving large files, such as images or videos, in MongoDB. When a file exceeds the BSON document size limit (16MB), GridFS splits the file into smaller chunks and stores each chunk as a separate document in a fs.chunks collection, with metadata stored in a fs.files collection. GridFS is ideal for storing large files and handling media storage within MongoDB.

**Example Use Case**: Storing and retrieving large media files such as videos or images.

* **Sharding**: Sharding is a method for distributing data across multiple machines. It allows MongoDB to scale horizontally by partitioning large datasets across multiple servers (shards). Each shard holds a subset of the data, and MongoDB distributes queries across all shards to balance the load.

**Example Use Case**: Distributing a large user database across multiple servers to handle high volumes of read and write operations.

In summary, **GridFS** is used for storing large files, while **Sharding** is used for distributing large datasets across multiple servers for scalability.

**10. Sharding**

### What is Sharding?

**Sharding** is a method of distributing data across multiple servers to handle large datasets and high throughput operations. It allows MongoDB to scale horizontally by splitting data into smaller, more manageable pieces called shards.

### Components of Sharding

1. **Shards**: Each shard holds a subset of the data. Shards are typically deployed as replica sets for high availability.
2. **mongos**: Acts as a query router, directing client requests to the appropriate shard.
3. **Config Servers**: Store metadata and configuration settings for the cluster.

### Query Routing in Sharding

The **mongos** instance routes queries to the appropriate shards based on the shard key. It determines which shards contain the relevant data and directs the query accordingly. This ensures efficient data retrieval and load balancing.

### Advantages and Disadvantages of Sharding

**Advantages**:

* **Scalability**: Handles large datasets and high throughput by distributing data.
* **Performance**: Improves read and write performance by spreading the load.
* **High Availability**: Each shard can be a replica set, ensuring data redundancy.

**Disadvantages**:

* **Complexity**: Increases the complexity of the database architecture.
* **Maintenance**: Requires careful management and monitoring.
* **Cost**: Can be more expensive due to the need for multiple servers.

### Sharding vs Replication

* **Sharding**: Distributes data across multiple servers to handle large datasets and high throughput. It focuses on horizontal scaling.
* **Replication**: Duplicates data across multiple servers to ensure high availability and fault tolerance. It focuses on data redundancy.

### Sharding Best Practices

1. **Choose the Right Shard Key**: Select a shard key that evenly distributes data.
2. **Monitor Performance**: Regularly monitor the performance of your sharded cluster.
3. **Plan for Growth**: Design your sharding strategy with future growth in mind.
4. **Use Indexes**: Ensure proper indexing to optimize query performance.

### CAP Theorem

The **CAP Theorem** states that a distributed database can only guarantee two out of three properties at the same time: **Consistency**, **Availability**, and **Partition Tolerance**. MongoDB prioritizes availability and partition tolerance.

### Capped Collections

**Capped Collections** are fixed-size collections that automatically overwrite the oldest data when they reach their size limit. [They are useful for logging and caching scenarios1](https://www.mongodb.com/docs/manual/sharding/).

### How to Create a Capped Collection

db.createCollection("myCappedCollection", { capped: true, size: 100000 });

This command creates a capped collection with a size limit of 100,000 bytes

### Sharding Disadvantages

* **Complexity**: Adds complexity to the database architecture.
* **Maintenance**: Requires careful management and monitoring.
* **Cost**: Can be more expensive due to the need for multiple servers.

**11. GridFS**

### What is GridFS?

**GridFS** is a specification in MongoDB for storing and retrieving large files, such as images, videos, and documents, that exceed the BSON document size limit of 16 MB. Instead of storing a file in a single document, GridFS divides the file into smaller chunks and stores each chunk as a separate document. This allows for efficient storage and retrieval of large files.

### Difference Between GridFS and Sharding

* **GridFS**: Used for storing large files by breaking them into smaller chunks. It is ideal for files that exceed the 16 MB limit and allows for partial file retrieval without loading the entire file into memory.
* **Sharding**: Distributes data across multiple servers to handle large datasets and high throughput. It improves performance and scalability by dividing the data into smaller, more manageable pieces.

### GridFS vs Traditional File Storage

* **GridFS**:
  + Stores files within MongoDB collections.
  + Allows for partial file retrieval, which is useful for streaming large files.
  + Automatically handles file metadata and synchronization across distributed systems.
* **Traditional File Storage**:
  + Stores files on the filesystem.
  + Requires additional mechanisms for metadata management and synchronization.
  + May have limitations on the number of files in a directory and does not inherently support partial file retrieval.

**12. Transactions and Batch Operations**

### Transactions in MongoDB

**Transactions** in MongoDB allow you to group multiple read and write operations into a single, atomic operation. This means that either all operations in the transaction succeed, or none do. Transactions ensure data consistency and are useful for complex operations that span multiple documents or collections.

### ACID Compliance

**ACID** stands for Atomicity, Consistency, Isolation, and Durability:

* **Atomicity**: All operations in a transaction are completed successfully or none are.
* **Consistency**: Transactions ensure that the database remains in a consistent state before and after the transaction.
* **Isolation**: Transactions are isolated from each other; intermediate states are not visible to other transactions.
* **Durability**: Once a transaction is committed, the changes are permanent, even in the event of a system failure.

### Batch Sizing

**Batch Sizing** in MongoDB controls the number of documents returned in each batch of a query response. Adjusting the batch size can optimize performance:

* **Large Batch Size**: Reduces the number of network round trips but uses more memory.
* **Small Batch Size**: Uses less memory but increases the number of network round trips.

### Upsert Operations

An **Upsert** operation in MongoDB is a combination of update and insert. If a document matching the query criteria exists, it updates the document. If no matching document is found, it inserts a new document. This is useful for ensuring that data is always up-to-date without needing separate insert and update logic.

### Use Cases for Transactions

Transactions are particularly useful in scenarios where multiple operations need to be executed as a single unit. Common use cases include:

* **Financial Transactions**: Ensuring that all steps in a financial transaction (like transferring money between accounts) are completed successfully.
* **Inventory Management**: Ensuring that inventory levels are updated correctly when processing orders.
* **Order Processing**: Ensuring that all parts of an order (like payment and inventory update) are completed together.

### Concurrency in Node.js with MongoDB

Concurrency in Node.js with MongoDB can be managed using various techniques:

* **Callbacks and Promises**: Use callbacks or promises to handle asynchronous operations.
* **Async/Await**: Simplifies asynchronous code, making it easier to read and maintain.
* **Transactions**: Use transactions to ensure that multiple operations are executed atomically, preventing race conditions and ensuring data consistency.

**13. Backup and Restore**

### MongoDB Backup and Restore Commands

* **Backup**: Use the mongodump command to create a backup of your MongoDB database.
  + mongodump --db mydatabase --out /backup/directory
* **Restore**: Use the mongorestore command to restore a MongoDB database from a backup.
  + mongorestore --db mydatabase /backup/directory/mydatabase

### Backup Best Practices

1. **Follow the 3-2-1 Rule**: Keep three copies of your data, two on different storage devices, and one off-site.
2. **Automate Backups**: Schedule regular backups to avoid forgetting.
3. **Test Your Backups**: Regularly test your backups to ensure they can be restored successfully.
4. **Use Encryption**: Encrypt your backups to protect sensitive data.
5. **Monitor Backup Processes**: Continuously monitor your backup processes to detect and resolve issues promptly.

### Restore Best Practices

1. **Document Your Restore Procedures**: Have clear, documented procedures for restoring data.
2. **Test Restores Regularly**: Regularly test your restore process to ensure it works as expected.
3. **Verify Data Integrity**: After restoring, verify the integrity and consistency of the data.
4. **Minimize Downtime**: Plan your restore process to minimize downtime and impact on users.
5. **Keep Backup Logs**: Maintain logs of backup and restore operations for auditing and troubleshooting.

**14. Security Best Practices**

**Authentication and Authorization in MongoDB**

* **Authentication**: Verifies the identity of a user or client. MongoDB supports various authentication mechanisms like SCRAM, x.509 certificates, LDAP, and Kerberos.
* **Authorization**: Determines what actions an authenticated user can perform. MongoDB uses Role-Based Access Control (RBAC) to manage permissions

**Role-Based Access Control (RBAC)**

* **RBAC**: Assigns roles to users, and each role has specific permissions. Roles can be built-in (like readWrite, dbAdmin) or custom-defined
* **Roles**: Control access to database resources and operations. Users can have multiple roles, and roles can inherit permissions from other roles2

**Encryption Best Practices**

1. **Encrypt Data at Rest**: Use MongoDB’s built-in encryption for data stored on disk. [This requires MongoDB Enterprise or MongoDB Atlas](https://www.mongodb.com/resources/products/capabilities/best-practices).
2. [**Encrypt Data in Transit**: Enable TLS/SSL to encrypt data as it travels over the network5](https://www.mongodb.com/products/capabilities/security/encryption).
3. [**Client-Side Field Level Encryption**: Encrypt sensitive fields on the client side before sending them to the server](https://www.mongodb.com/docs/manual/core/authentication/).
4. [**Key Management**: Use a secure Key Management System (KMS) to store and manage encryption keys](https://www.percona.com/blog/securing-your-mongodb-database-essential-best-practices/).

**Secure Network Configuration**

* [**IP Binding**: Bind MongoDB to specific IP addresses to limit access to trusted networks](https://www.mongodb.com/docs/manual/core/authentication/)
* [**Firewalls**: Use firewalls to control incoming and outgoing traffic to MongoDB instances](https://www.mongodb.com/docs/manual/core/authentication/).
* [**VPNs**: Use Virtual Private Networks (VPNs) to secure connections between clients and MongoDB servers](https://www.mongodb.com/docs/manual/core/security-hardening/)
* [**Disable IP Forwarding**: Prevent servers from forwarding packets to other systems](https://www.mongodb.com/docs/manual/core/security-hardening/)

**Auditing and Logging in MongoDB**

* **Auditing**: Tracks and logs database events like user authentication, command execution, and configuration changes. [This helps in monitoring and analyzing database activity](https://www.mongodb.com/docs/manual/core/auditing/)
* **Audit Logs**: Can be written to the console, syslog, JSON files, or BSON files. [Configure audit filters to capture specific events](https://www.mongodb.com/docs/manual/core/authentication/)
* [**Logging**: Regularly review audit logs to detect and respond to suspicious activities](https://www.mongodb.com/docs/manual/core/authentication/)

**15. Performance Tuning**

### Indexing for Performance

**Indexing** improves database performance by creating a data structure that allows for faster retrieval of records. Think of it like an index in a book, which helps you quickly find the information you need without reading every page. In MongoDB, indexes can be created on fields that are frequently queried to speed up search operations.

### Query Optimization

**Query Optimization** involves refining queries to reduce execution time and resource consumption. This can be achieved by:

* **Using indexes**: Ensure queries use indexes to avoid full collection scans.
* **Avoiding unnecessary data retrieval**: Only fetch the fields you need.
* **Optimizing joins and aggregations**: Simplify complex queries and use efficient join operations.

### Caching Strategies

**Caching** stores frequently accessed data in a temporary storage area to reduce access time. Common caching strategies include:

* **Cache-Aside**: The application checks the cache first before querying the database.
* **Read-Through**: The cache automatically loads data from the database on a cache miss.
* **Write-Through**: Data is written to the cache and the database simultaneously.
* **Write-Back**: Data is written to the cache first and then asynchronously to the database.

### Load Balancing

**Load Balancing** distributes incoming network traffic across multiple servers to ensure no single server becomes overwhelmed. This improves application performance and reliability by:

* **Distributing traffic**: Spreading requests evenly across servers.
* **Failover**: Redirecting traffic to healthy servers if one fails.
* **Scalability**: Adding or removing servers based on demand

### Performance Tuning Best Practices

1. **Keep Statistics Up to Date**: Ensure database statistics are current to generate optimal execution plans.
2. **Avoid Leading Wildcards**: Leading wildcards in queries force full table scans, which are slow.
3. **Use Constraints**: Constraints help the database optimizer create better execution plans.
4. \*\*Avoid SELECT \*\*\*: Only retrieve the fields you need to reduce data transfer and processing time.
5. **Monitor and Analyze**: Regularly monitor performance metrics and analyze slow queries to identify bottlenecks

**16. MongoDB Atlas**

### Overview of MongoDB Atlas

**MongoDB Atlas** is a fully managed, multi-cloud database service that simplifies deploying, managing, and scaling MongoDB databases. It allows you to build resilient and performant global applications on the cloud providers of your choice, such as AWS, Azure, and Google Cloud.

### Setting Up MongoDB Atlas

1. **Sign Up**: Create an account on the MongoDB Atlas website.
2. **Create a Cluster**: Choose a cloud provider and region, and create a new cluster. You can start with a free tier cluster for development and testing.
3. **Configure Access**: Set up a database user and add your IP address to the access list.
4. **Connect to Your Cluster**: Use the connection string provided by Atlas to connect your application to the database.

### Atlas Clustering

**Atlas Clustering** involves creating clusters that can be either replica sets or sharded clusters:

* **Replica Sets**: Provide high availability and redundancy by replicating data across multiple nodes.
* **Sharded Clusters**: Distribute data across multiple shards to handle large datasets and high throughput.

### Security Features in Atlas

MongoDB Atlas comes with several built-in security features to protect your data:

1. **Encryption in Transit**: Uses TLS/SSL to encrypt data as it travels over the network.
2. **Encryption at Rest**: Encrypts data stored on disk to protect it from unauthorized access.
3. **IP Access List**: Restricts database access to specified IP addresses.
4. **User Authentication and Authorization**: Uses Role-Based Access Control (RBAC) to manage permissions.
5. **Network Isolation**: Supports Virtual Private Cloud (VPC) peering and private endpoints for secure network configurations.
6. **Auditing**: Tracks and logs database events for monitoring and compliance.

**17. Document Validation**

### Schema Validation in MongoDB

**Schema Validation** in MongoDB allows you to define rules for the structure of documents in a collection. This ensures that all documents adhere to a specified format, which helps maintain data integrity and consistency. You can specify validation rules using JSON Schema syntax, which includes constraints like data types, required fields, and value ranges.

### Custom Validation Rules

You can create custom validation rules using the $jsonSchema operator. Here’s an example of setting up a validation rule for a collection:

db.createCollection("students", {

validator: {

$jsonSchema: {

bsonType: "object",

required: ["name", "age", "gpa"],

properties: {

name: {

bsonType: "string",

description: "must be a string and is required"

},

age: {

bsonType: "int",

minimum: 0,

description: "must be an integer greater than or equal to 0"

},

gpa: {

bsonType: "double",

minimum: 0,

maximum: 4,

description: "must be a double between 0 and 4"

}

}

}

}

});

This rule ensures that every document in the students collection has a name (string), age (integer), and gpa (double between 0 and 4).

### Validation Best Practices

1. **Start Simple**: Begin with basic validation rules and gradually add complexity as needed.
2. **Use Descriptive Messages**: Include descriptions in your validation rules to provide clear error messages.
3. **Test Regularly**: Regularly test your validation rules to ensure they work as expected.
4. **Combine with Application-Level Validation**: While MongoDB’s validation provides a safety net, also validate data at the application level for more control.
5. **Monitor and Adjust**: Continuously monitor the effectiveness of your validation rules and adjust them based on your application’s needs.

**18. Miscellaneous Topics**

### CAP Theorem

The **CAP Theorem** states that in a distributed database system, you can only achieve two out of the following three guarantees at the same time:

* **Consistency**: Every read receives the most recent write.
* **Availability**: Every request receives a response, even if it’s not the most recent.
* **Partition Tolerance**: The system continues to operate despite network partitions.

### TTL (Time to Live)

**TTL (Time to Live)** is a mechanism that limits the lifespan of data. In MongoDB, TTL indexes are used to automatically delete documents after a specified period. This is useful for managing data that only needs to be retained for a certain amount of time, like session data or logs.

### Data Redundancy

**Data Redundancy** refers to the practice of storing the same piece of data in multiple places. This can be intentional for backup and recovery purposes or accidental due to inefficient data management. While redundancy can improve data availability and fault tolerance, it can also lead to data inconsistency and increased storage costs if not managed properly.

### Clustered Collections

**Clustered Collections** in MongoDB store documents ordered by a clustered index key. This means that the documents are physically stored in the order of the index key, which can improve query performance for range queries and equality comparisons on the clustered index key.

### Materialized Views

A **Materialized View** is a database object that contains the results of a query. Unlike regular views, which are virtual and recomputed each time they are accessed, materialized views store the query results physically. This can significantly improve query performance, especially for complex queries that are frequently executed.

.

### Decrement Operations

**Decrement Operations** in MongoDB are used to decrease the value of a field. This can be done using the $inc operator with a negative value. For example:

db.collection.updateOne(

{ \_id: 1 },

{ $inc: { count: -1 } }

);

This command decreases the count field by 1.

* Alternatives to MongoDB

### Cassandra

**Apache Cassandra** is a highly scalable, distributed NoSQL database designed to handle large amounts of data across many commodity servers without a single point of failure. It is known for its high availability, fault tolerance, and linear scalability.

### Redis

**Redis** (Remote Dictionary Server) is an in-memory data structure store used as a database, cache, and message broker. It supports various data structures such as strings, hashes, lists, sets, and more. Redis is known for its high performance and low latency.

### DynamoDB

**Amazon DynamoDB** is a fully managed NoSQL database service provided by AWS. It offers fast and predictable performance with seamless scalability. DynamoDB is designed for applications that require consistent, single-digit millisecond latency at any scale.

### HBase

**Apache HBase** is an open-source, distributed, scalable, and NoSQL database modeled after Google’s Bigtable. It is designed to handle large amounts of sparse data and is built on top of the Hadoop Distributed File System (HDFS). HBase is known for its strong consistency and random, real-time read/write access.

### OrientDB

**OrientDB** is a multi-model NoSQL database that supports graph, document, key-value, and object models. It is designed to be highly scalable and efficient, combining the flexibility of document databases with the power of graph databases.

Scaling in MongoDB is essential for handling increasing data volumes, user traffic, and processing demands. There are two main methods for scaling MongoDB: **vertical scaling** and **horizontal scaling**.

### Vertical Scaling (Scaling Up)

* **Definition**: Increasing the capacity of a single server by adding more resources (CPU, RAM, storage).
* **Use Case**: Suitable for applications with moderate growth where a single server can handle the increased load.
* **Example**: Upgrading your server from 16GB RAM to 32GB RAM to handle more queries and data.

### Horizontal Scaling (Scaling Out)

* **Definition**: Adding more servers to distribute the load and data across multiple machines.
* **Use Case**: Ideal for applications with significant growth, requiring more resources than a single server can provide.
* **Techniques**:
  + **Replication**: Creating copies of the database on multiple servers to ensure high availability and fault tolerance.
  + **Sharding**: Distributing data across multiple servers (shards) to balance the load and improve performance.