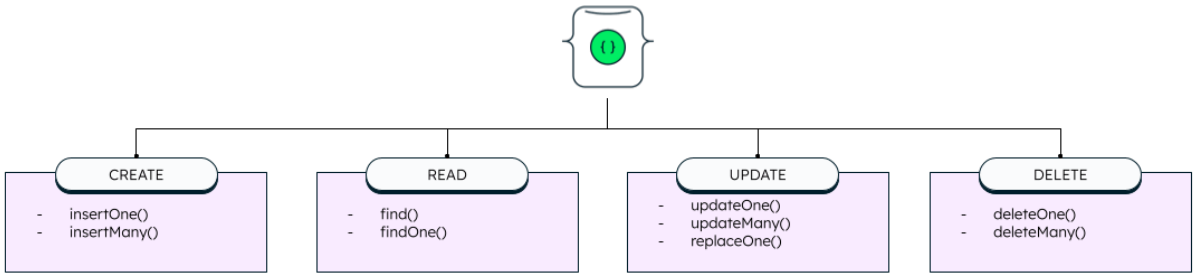
I. CRUD Operations

CRUD Operations (Create, Read, Update, and Delete) are the basic set of operations that allow users to interact with the MongoDB server.

To use MongoDB we need to interact with the MongoDB server to perform certain operations like entering new data into the application, updating data into the application, deleting data from the application, and reading the application data.

MongoDB CRUD operations at a glance



➢ Create operation

For MongoDB CRUD, if the specified collection doesn't exist, the create operation will create the collection when it's executed. Create operations in MongoDB target a single collection, not multiple collections. Insert operations in MongoDB are atomic on a single document level.

MongoDB provides two different create operations that you can use to insert documents into a collection:

- db.collection.insertOne()

- db.collection.insertMany()

insertOne()

insertOne() allows you to insert one document into the collection. For this example, we're going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB. We then provide the information we want to insert in the form of key-value pairs, establishing the schema. Below is the example:

```
db.RecordsDB.insertOne({
   name: "Marsh",
   age: "6 years",
   species: "Dog",
   ownerAddress: "380 W. Fir Ave",
   chipped: true
})
```

insertMany()

It's possible to insert multiple items at one time by calling the *insertMany()* method on the desired collection. In this case, we pass multiple items into our chosen collection (*RecordsDB*) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries. This is commonly referred to as a nested method.

```
db.RecordsDB.insertMany([{
   name: "Marsh",
   age: "6 years",
   species: "Dog",
   ownerAddress: "380 W. Fir Ave",
   chipped: true},
    {name: "Kitana",
    age: "4 years",
    species: "Cat",
    ownerAddress: "521 E. Cortland",
    chipped: true}])
```

- **Read operations**

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on

the available query filters. Query modifiers may also be used to change how many results are returned.

MongoDB has two methods of reading documents from a collection:

- db.collection.find()
- db.collection.findOne()

find()

To get all the documents from a collection, we can simply use the *find()* method on our chosen collection. Executing just the *find()* method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
```

- **Update operations**

Like create operations, update operations operate on a single collection, and they are atomic at a single document level. An update operation takes filters and criteria to select the documents you want to update.

You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- db.collection.updateOne()
- db.collection.updateMany()
- db.collection.replaceOne()

updateOne()

We can update a currently existing record and change a single document with an update operation. To do this, we use the *updateOne()* method on a chosen collection, which here is "RecordsDB." To update a document, we provide the method with two arguments: an update filter and an update action.

The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter. Then, we use the "$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne({name: "Marsh"}, {$set:{ownerAddress: "451 W. Coffee St. A204"}})
```

updateMany()

updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({species:"Dog"}, {$set: {age: "5"}})
```

replaceOne()

The *replaceOne()* method replaces a single document in the specified collection. *replaceOne()* replaces the entire document, meaning fields in the old document not contained in the new one will be lost.

```
db.RecordsDB.replaceOne({name: "Kevin"}, {name: "Maki"})
```

- **Delete operations**

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

deleteOne() removes a document from a specified collection on the MongoDB server. A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({name:"Maki"})
```

deleteMany()

deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation. A list is passed into the method and the individual items are defined with filter criteria as in *deleteOne()*.

```
db.RecordsDB.deleteMany({species:"Dog"})
```

II. Operators in MongoDB

- **MongoDB Query Operators**

  There are many query operators that can be used to compare and reference document fields.

  Comparison:

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

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

  Logical:

  The following operators can logically compare multiple queries.
- $and: Returns documents where both queries match
- $or: Returns documents where either query matches
- $nor: Returns documents where both queries fail to match
- $not: Returns documents where the query does not match

  Evaluation:

  The following operators assist in evaluating documents.
- $regex: Allows the use of regular expressions when evaluating field values
- $text: Performs a text search
- $where: Uses a JavaScript expression to match documents

➢ **MongoDB Update Operators**

There are many update operators that can be used during document updates.

Fields:

The following operators can be used to update fields:

- $currentDate: Sets the field value to the current date
- $inc: Increments the field value
- $rename: Renames the field
- $set: Sets the value of a field
- $unset: Removes the field from the document

Array:

The following operators assist with updating arrays.

- $addToSet: Adds distinct elements to an array
- $pop: Removes the first or last element of an array
- $pull: Removes all elements from an array that match the query
- $push: Adds an element to an array

➢ **Aggregation Operators**

- **Arithmetic Expression Operators**

Arithmetic expressions perform mathematic operations on numbers. Some arithmetic expressions can also support date arithmetic.

Name	Description
$abs	Returns the absolute value of a number.
$add	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
$ceil	Returns the smallest integer greater than or equal to the specified number.
$divide	Returns the result of dividing the first number by the second. Accepts two argument expressions.
$exp	Raises *e* to the specified exponent.
$floor	Returns the largest integer less than or equal to the specified number.
$ln	Calculates the natural log of a number.
$log	Calculates the log of a number in the specified base.
$log10	Calculates the log base 10 of a number.
$mod	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
$multiply	Multiplies numbers to return the product. Accepts any number of argument expressions.
$pow	Raises a number to the specified exponent.
$round	Rounds a number to to a whole integer *or* to a specified decimal place.
$sqrt	Calculates the square root.
$subtract	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.
$trunc	Truncates a number to a whole integer *or* to a specified decimal place.

- **Bitwise Operators**

Name	Description
$bitAnd	Returns the result of a bitwise `and` operation on an array of `int` or `long` values. *New in version 6.3*.
$bitNot	Returns the result of a bitwise `not` operation on a single argument or an array that contains a single `int` or `long` value. *New in version 6.3*.
$bitOr	Returns the result of a bitwise `or` operation on an array of `int` or `long` values. *New in version 6.3*.

$bitXor	Returns the result of a bitwise xor (exclusive or) operation on an array of int and long values. *New in version 6.3*.

- Boolean Expression Operators

Boolean expressions evaluate their argument expressions as booleans and return a boolean as the result.

In addition to the false boolean value, Boolean expression evaluates as false the following: null, 0, and undefined values. The Boolean expression evaluates all other values as true, including non-zero numeric values and arrays.

Name	Description
$and	Returns true only when *all* its expressions evaluate to true. Accepts any number of argument expressions.
$not	Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.
$or	Returns true when *any* of its expressions evaluates to true. Accepts any number of argument expressions.

- **Conditional Expression Operators**

Name	Description
$cond	A ternary operator that evaluates one expression, and depending on the result, returns the value of one of the other two expressions. Accepts either three expressions in an ordered list or three named parameters.
$ifNull	Returns either the non-null result of the first expression or the result of the second expression if the first expression results in a null result. Null result encompasses instances of undefined values or missing fields. Accepts two expressions as arguments. The result of the second expression can be null.
$switch	Evaluates a series of case expressions. When it finds an expression which evaluates to true, $switch executes a specified expression and breaks out of the control flow.

- **Custom Aggregation Expression Operators**

Name	Description
$accumulator	Defines a custom accumulator function.
$function	Defines a custom function.

III. Modifiers in MongoDB

Query modifiers in MongoDB are essential tools that adjust the behavior of queries to achieve various objectives such as shaping results, enhancing performance and efficiently retrieving documents from the database. These operators allow developers to manipulate query results for tasks like sorting, projection and making queries more effective and readable.

Modifiers

Below are the some modifier which are frequently used in the MongoDB to perform operations on database.

Name	Description
$comment	Used to add a comment to the query
$explain	Used to report documents with an execution plan
$hint	Used to specify the index for a query
max()	Used to specify the upper limit for the index of a query
min()	Used to specify a lower limit for the index of a query
$maxTimeMS	Used to specify the maximum time for query execution in milliseconds
$orderby	Deprecated, used for sorting documents
$query	Used to specify conditions to request a document
$returnKey	Used to provide the index key from the result
$showDiskLoc	Deprecated, used to show disk location

Examples

1. $comment in MongoDB

$comment is used to add a comment to the query. Comments are not stored in the database. If we want comments to be stored , add the comment as field and value in the document. $comment can be used as follows

Syntax:

```
db.collection.find( { condition }).comment( add_comment_related_to_condition )
or
db.collectionj.find(  {  condition } ,$comment : "comment"   )
```

Query:

```
//collection used for example.
 db.Digits.insertMany([{value:1},{value:12},{value:21},{value:27}]);
the db.Digits.find({ value: { $mod: [3, 1] } }).comment("Number not divisible by 3");
```

2. min() and max() in MongoDB

min() and max() are used to specify lower and upper limits respectively for the index in a query.

Syntax for min():

db.collection.find(condition).min({ field: lower limit }).hint({index})

Syntax for max():

db.collection.find({condition}).max({ field: upper limit }).hint(index);

Example: Use min() and max() on Digits collection

Query:

```
db.Digits.find( { value : {$ gt :10}}  ).min( { value:5}).hint( { value: 1});
db.Digits.find( { value : {$ lt :25}}  ).max( { value:20}).hint( { value: 1});
```

3. $maxTimeMS in MongoDB

$maxTimeMS is used to specify the maximum time in which the query should be executed. $maxTimeMS operator is deprecated instead of maxTimeMs() is used. Time is specified in millisecond with integral value and it should be in the range of [0 , 2147483647].

Syntax:

db.collection_name.find().maxTimeMs(mention maximum time);

Query:

db.Digits.find().maxTimeMS(100);

4. $orderby in MongoDB

$orderby is deprecated, sort() method performs the same operation as $orderby. sort() carries out sorting of the documents in a specific order.Sort direction is specified by 1 and -1.1 for ascending order and -1 for descending.

Syntax:

For ascending order
db.collection_name.find().sort({field: 1});
for descending order
db.collection_name.find().sort({ field : -1})

Query:

db.Digits.find().sort({ values: -1})

IV. Indexing in MongoDB

Indexing in **MongoDB** is a crucial feature that enhances query processing efficiency. Without indexing, MongoDB must scan every document in a collection to retrieve the **matching** documents and leading to **slower query performance**. **Indexes** are special data structures that store information about

the documents in a way that makes it easier for MongoDB to quickly locate the right data.

In this article, We will learn about **Indexing in MongoDB** by understanding **how to create, drop and get an Index** with the help of examples and so on.

- **Indexing in MongoDB**

- MongoDB uses **indexing** to make the query processing more efficient.
- If there is no indexing, then MongoDB must scan every document in the collection and retrieve only those documents that match the query.
- Indexes are special data structures that store some information related to the documents such that it becomes easy for MongoDB to find the right data file.
- The indexes are ordered by the value of the field specified in the index.

- **Creating an Index**

  MongoDB provides a method called createIndex() that allows users to create an index.

  Syntax:

  db.COLLECTION_NAME.createIndex({KEY:1})

  The key determines the field on the basis of which we want to create an index and 1 (or -1) determines the order in which these indexes will be arranged(ascending or descending).

  Example of Indexing in MongoDB

  db.mycol.createIndex({"age":1})

- **Drop an index**

  In order to drop an index, MongoDB provides the dropIndex() method.

  Syntax:

  db.NAME_OF_COLLECTION.dropIndex({KEY:1})

  Syntax:

  db.NAME_OF_COLLECTION.dropIndexes({KEY1:1, KEY2: 1})

V. Single index, Finding index.

A **single field index** in MongoDB is an index created on a single field of a document. It improves query performance by allowing MongoDB to quickly find documents based on that field instead of scanning the entire collection.

1. Creating a Single Field Index

To create an index on a single field, use the createIndex() method.

Syntax:

db.collection.createIndex({ fieldName: 1 }) // 1 for ascending, -1 for descending

Example:

{ "_id": 1, "name": "Alice", "age": 30, "city": "New York" }
{ "_id": 2, "name": "Bob", "age": 25, "city": "San Francisco" }
{ "_id": 3, "name": "Charlie", "age": 35, "city": "Chicago" }

Now, we create an index on the name field:

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

2. Sorting with an Index

Indexes also improve sorting performance.

Example:
Creating an index on age in descending order:
db.users.createIndex({ age: -1 })
Now, sorting queries like:
db.users.find().sort({ age: -1 })

3. Viewing Existing Indexes
To check which indexes exist in a collection:
db.users.getIndexes()

4. Dropping an Index
If an index is no longer needed, you can remove it to free up resources.
Example:
To remove the index on name:
db.users.dropIndex({ name: 1 })

5. When to Use Single Field Index
✅ When frequently querying or sorting by a single field.
✅ When filtering a large collection on a specific field.
❌ Not useful for queries involving multiple fields (use a **compound index** instead).

VI. Multikey Index

A **Multikey Index** in MongoDB is used to index fields that contain **arrays**. This allows efficient querying of documents that have one or more values in an array field.

1. Why Use a Multikey Index?

MongoDB automatically creates a **Multikey Index** when you create an index on an **array field**. This improves performance when searching for elements inside an array.

2. Creating a Multikey Index

You don't need to specify that an index is **Multikey**—MongoDB automatically detects when an indexed field contains an array.

Example:

Sample Data (products collection)

```
{ "_id": 1, "name": "Laptop", "tags": ["electronics", "computers", "gadgets"] }
{ "_id": 2, "name": "Phone", "tags": ["electronics", "mobile", "gadgets"] }
{ "_id": 3, "name": "Shoes", "tags": ["fashion", "footwear"] }
```

Creating an Index on the tags Array

```
db.products.createIndex({ tags: 1 })
```

3. Querying Using a Multikey Index

Once the index is created, queries on array fields become much faster.

Find documents with "electronics" in tags:
db.products.find({ tags: "electronics" })

This query will use the **Multikey Index** and be much faster.

Find documents with multiple values in tags:
db.products.find({ tags: { $all: ["electronics", "gadgets"] } })

The $all operator ensures that both values are present in the array.

4. Key Takeaways

✅ Used for **arrays** automatically.
✅ Improves performance when querying inside arrays.
✅ Sorting is possible **only** if the Multikey Index is the **only array field** in the index.
✅ Compound indexes **cannot** contain **multiple array fields**.

VII. Aggregation Framework

Aggregation in **MongoDB** is a powerful framework that allows developers to perform complex **data transformations**, computations and analysis on collections of documents. By utilizing the **aggregation** pipeline, users can efficiently **group**, **filter**, **sort**, **reshape**, and perform calculations on data to generate meaningful insights.

In this article, We will explain **MongoDBAggregation** in detail from basic concepts to advanced use cases, ensuring we have a thorough understanding of its capabilities, covering various aspects related to **MongoDB Aggregation**.

What is Aggregation?

MongoDB Aggregation is a database process that allows us to perform complex **data transformations** and **computations** on collections of documents or rows. It enables us to **group**, **filter**, and **manipulate** data to produce summarized results. **MongoDBAggregation** is typically carried out using the aggregation pipeline which is a framework for **data aggregation** modeled on the concept of data processing pipelines.

Each stage of the pipeline transforms the documents as they pass through it and allows for operations like **filtering**, **grouping**, **sorting**, **reshaping** and performing calculations on the data. MongoDB supports two primary aggregation approaches:

1. Single Purpose Aggregation

Single-purpose aggregation methods are designed for simple **analytical queries**. It is used when we need simple access to document like counting the number of documents or for finding all **distinct values** in a document. It simply provides the

access to the **common aggregation** which provides straightforward aggregation functions like:

- **count()** – Returns the number of documents in a collection.
- **distinct()**– Retrieves unique values for a specified field.
- **estimatedDocumentCount()** – Provides an estimated count of documents.

Syntax:

```
db.collection.aggregate([
  { stage1 },
  { stage2 },
  { stage3 }
])
```

2. Mongodb Aggregation Pipeline

Mongodb Aggregation Pipeline consist of stages and each stage transforms the document. It is a **multi-stage pipeline** and in each state the documents are taken as input to produce the resultant set of documents. In the next stage (ID available) the resultant documents are taken as input to produce output, this process continues till the last stage. Each stage filters, modifies, or computes on documents before passing them to the next stage.

The **basic pipeline stages** are defined below:

- filters that will operate like queries.
- the document transformation that modifies the resultant document.
- provide pipeline tools for **grouping and sorting documents.**

VIII. Pipeline Operations

The **Aggregation Pipeline** in MongoDB is a **framework** that processes data in multiple **stages**. Each stage applies a transformation to the documents and passes the result to the next stage.

1. Basic Syntax

```
db.collection.aggregate([
  { stage1 },
  { stage2 },
  { stage3 }
])
```

Each **stage** processes data and passes the output to the next stage.

2. Common Pipeline Stages (Operations)

- **$match – Filter Documents (Like find())**

Filters documents based on conditions.

Syntax:

```
{ $match: { field: value } }
```

Example: Find users older than 25

```
db.users.aggregate([
  { $match: { age: { $gt: 25 } } }
])
```

✅ **Tip**: $match early in the pipeline **improves performance**.

🔹 **$group – Group Documents**

Groups documents by a field and applies aggregation functions.

Syntax:

{ $group: { _id: "$field", newField: { $operator: "$field" } } }

Example: Count users by city

```
db.users.aggregate([
  { $group: { _id: "$city", totalUsers: { $sum: 1 } } }
])
```

Example: Find average age of users in each city

```
db.users.aggregate([
  { $group: { _id: "$city", avgAge: { $avg: "$age" } } }
])
```

🔹 **$sort – Sort Documents**

Sorts in **ascending (1)** or **descending (-1)** order.

Syntax:

{ $sort: { field: 1 or -1 } }

Example: Sort users by age (descending)

```
db.users.aggregate([
  { $sort: { age: -1 } }
])
```