

Chapter 15: MongoDB CRUD & Querying

15.1 insertOne()

- * `insertOne()` is used to insert a **single document** into a collection.
- * If the document is inserted successfully, it returns a **result object**.
- * If `_id` is not provided, MongoDB automatically generates a unique `_id` for the document.
- * It is the most basic method to add data into a collection.

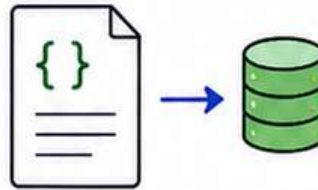


Syntax:

```
db.collection.insertOne( { document } )
```

Example:

```
db.students.insertOne(  
  {  
    name: "Bhavitha",  
    age: 22,  
    course: "BTech CSE",  
    city: "Hyderabad"  
  }  
)
```



Output:

```
{  
  "acknowledged": true,  
  "insertedId": ObjectId(" 663d9f2b8f4a1a5b6c7d8e91")  
}
```

`insertedId` is the unique id generated for the inserted document.

Key Points:

- ✓ `insertOne()` inserts a single document into the collection.
- ✓ If `_id` is not specified, MongoDB generates it automatically.
- ✓ It returns a result object with **acknowledged** status and **insertedId**.
- ✓ It is the simplest method to create data in MongoDB.



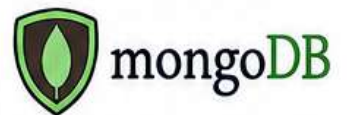
Common Use Case:

- Used when we want to add **one new record**, such as adding a new student, a new user, or a new product.



15.2 insertMany()

- * insertMany() is used to insert **multiple documents** into a collection.
- * It accepts an **array** of documents and inserts them in a single operation.
- * It is **more efficient** than inserting documents one by one.
- * If **_id** is not provided for any document, MongoDB automatically generates a unique **_id** for each document.
- * It returns a **result object** containing the inserted ids.

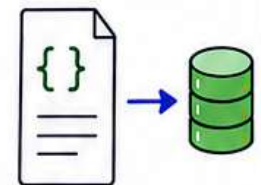


Syntax:

```
db.collection.insertMany( { document1, document2, ... } )
```

Example:

```
db.students.insertMany([
  { name: "Bhavitha", age: 22, course: "BTech CSE", city: "Hyderabad" },
  { name: "Anjali", age: 21, course: "BTech IT", city: "Vizag" },
  { name: "Ravi", age: 23, course: "BTech ECE", city: "Chennai" }
])
```



Output:

```
{
  "acknowledged": true,
  "insertedIds": {
    "0": ObjectId("663da0c1b8f4a1a5b6c7d8e9"),
    "1": ObjectId("663da0c1b8f4a1a5b6c7d8ea"),
    "2": ObjectId("663da0c1b8f4a1a5b6c7d8eb")
  }
}
```

insertedIds contains the ids of all documents inserted in the collection.

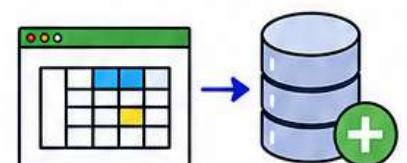
Key Points:

- ✓ insertMany() inserts multiple documents in one operation.
- ✓ It accepts an **array** of documents.
- ✓ **_id** is automatically generated for documents if not provided.
- ✓ Returns a result object with **insertedIds**.
- ✓ More efficient than inserting documents one by one.



Common Use Case:

- Used when we need to add multiple records at once, such as importing data, bulk inserts, or seeding data.



15.3 find()

- * `find()` is used to **retrieve documents** from a collection.
- * It returns a **cursor** to the matching documents.
- * If no documents match the query, it returns an **empty cursor**.
- * We can specify a **filter** to find specific documents.
- * It is one of the most commonly used methods in MongoDB.



Syntax:

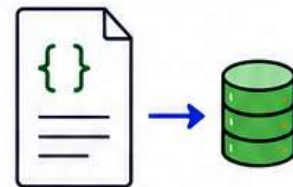
```
db.collection.find( filter , projection )
```

Example:

```
// Find all documents in the collection
db.students.find()

// Find documents where course is "BTech CSE"
db.students.find( { course: "BTech CSE" } )

// Find documents where age is greater than 20
db.students.find( { age: { $gt: 20 } } )
```



Output:

```
[
  { "_id": ObjectId("663da0c1b8f4a1a5b6c7d8e9"), "name": "Bhavitha",
    "age": 22, "course": "BTech CSE", "city": "Hyderabad" },
  { "_id": ObjectId("663da0c1b8f4a1a5b6c7d8ea"), "name": "Anjali",
    "age": 21, "course": "BTech IT", "city": "Vizag" },
  { "_id": ObjectId("663da0c1b8f4a1a5b6c7d8eb"), "name": "Ravi",
    "age": 23, "course": "BTech ECE", "city": "Chennai" }
]
```

Returns a list (**cursor**) of documents that match the query.

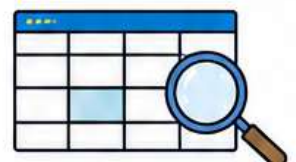
Key Points:

- ✓ `find()` retrieves documents from a collection based on the given **filter**.
- ✓ It returns a **cursor** to the matching documents.
- ✓ If **filter** is not provided, all documents are returned.
- ✓ We can use **projection** to specify which fields to include or exclude.
- ✓ The cursor can be iterated to access the documents.



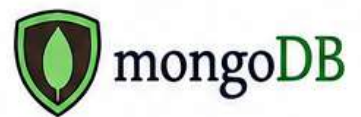
Common Use Case:

- Used when we want to read and display data from the database, such as showing all users, searching records, or filtering data.



15.4 findOne()

- * findOne() is used to retrieve a **single document** from a collection.
- * It returns the **first document** that matches the query.
- * If no document matches the query, it returns **null**.
- * If no filter is provided, it returns the **first document** in the collection.
- * It is useful when we expect only one matching document.



Syntax:

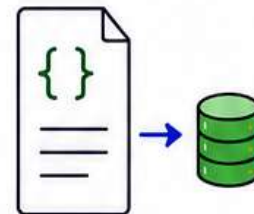
```
db.collection.findOne( filter , projection )
```

Example:

```
// Get the first document in the collection
db.students.findOne()

// Find the first document where course is "BTech CSE"
db.students.findOne( { course: "BTech CSE" } )

// Find the first document where age is greater than 20
db.students.findOne( { age: { $gt: 20 } } )
```



Output:

```
{
  "_id" : ObjectId("663da0c1b8f4a1a5b6c7d8e9"),
  "name" : "Bhavitha",
  "age" : 22,
  "course" : "BTech CSE",
  "city" : "Hyderabad"
}
```

Returns a **single** document (the first match) or **null** if no match is found.

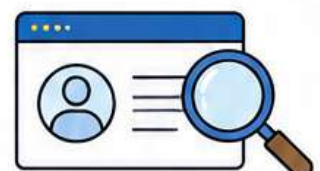
Key Points:

- ✓ **findOne()** returns only one document.
- ✓ It returns the **first document** that matches the query.
- ✓ If no filter is given, it returns the **first document** in the collection.
- ✓ If no document matches, it returns **null**.
- ✓ It is useful when we need a single record, not a list.



Common Use Case:

- Used when we need to get a single record, such as finding a user by email, username, or ID.



15.5 updateOne()

- * `updateOne()` is used to update a **single document** in a collection.
- * It updates the **first document** that matches the filter.
- * If no document matches the filter, **no update** is performed.
- * It uses update operators like **`$set`**, **`$inc`**, **`$unset`**, etc.
- * It returns a **result object** with details about the update operation.



Syntax:

```
db.collection.updateOne( filter , update , options )
```

Example:

```
// Update the first document where name is "Bhavitha", set city to "Bangalore"
db.students.updateOne(
  { name: "Bhavitha" },
  { $set: { city: "Bangalore" } }
)
```



Output:

```
{
  "acknowledged" : true,
  "matchedCount" : 1,
  "modifiedCount" : 1,
  "upsertedCount" : 0
}
```

matchedCount - number of documents matched by the filter.
modifiedCount - number of documents actually updated.
upsertedCount - number of new documents inserted (0 in this case).

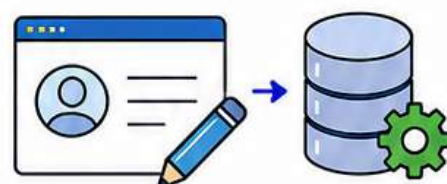
Key Points:

- ✓ `updateOne()` updates only **one document** that matches the **filter**.
- ✓ It updates the **first matching** document.
- ✓ If no document matches, **matchedCount** will be **0**.
- ✓ Use update operators like **`$set`**, **`$inc`**, **`$unset`**, **`$push`**, etc.
- ✓ The **`_id`** field should not be updated.



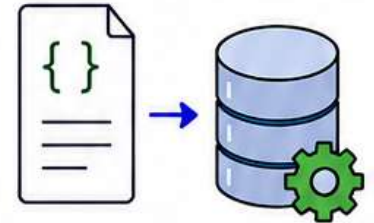
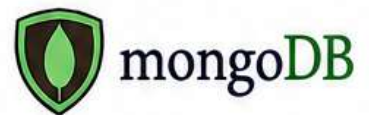
Common Use Case:

- Used when we need to update a single record, such as updating user profile, status, or any specific field.



15.6 updateMany()

- * `updateMany()` is used to update **multiple documents** in a collection.
- * It updates **all documents** that match the filter.
- * If no documents match the filter, **no update** is performed.
- * It uses update operators like **`$set`**, **`$inc`**, **`$unset`**, etc.
- * It returns a **result object** with details about the update operation.



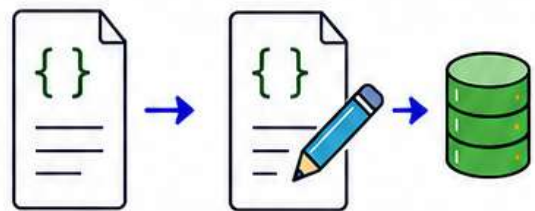
Syntax:

```
db.collection.updateMany( filter , update , options )
```

Example:

```
// Update all documents where course is "BTech CSE", set city to "Hyderabad"
```

```
db.students.updateMany(  
  { course: "BTech CSE" },  
  { $set: { city: "Hyderabad" } }  
)
```



Output:

```
{  
  "acknowledged" : true,  
  "matchedCount" : 3,  
  "modifiedCount" : 3,  
  "upsertedCount" : 0  
}
```

matchedCount - number of documents matched by the filter.
modifiedCount - number of documents actually updated.
upsertedCount - number of new documents inserted (0 in this case).

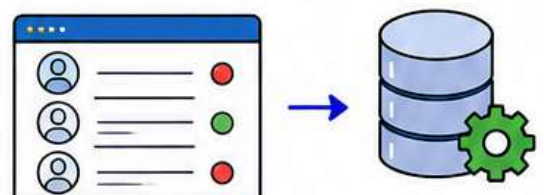
Key Points:

- ✓ `updateMany()` updates **all documents** that match the filter.
- ✓ If no documents match, **matchedCount** will be 0.
- ✓ It can modify **multiple documents** in a single operation.
- ✓ Use update operators like **`$set`**, **`$inc`**, **`$unset`**, **`$push`**, etc.
- ✓ The **`_id`** field should not be updated.



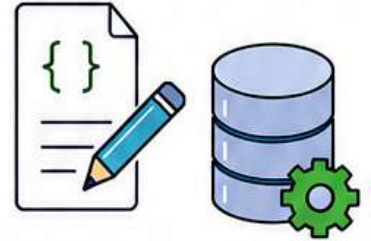
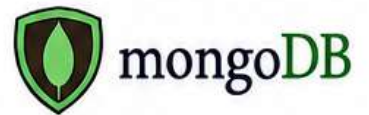
Common Use Case:

- Used when we need to update multiple records that share a common condition, such as updating status for all inactive users.



15.7 replaceOne()

- * `replaceOne()` is used to replace a **single document** in a collection.
- * It replaces the **first document** that matches the filter.
- * The **entire document** is replaced with a **new document**.
- * If no document matches the filter, **no replacement** is performed.
- * It returns a **result object** with details about the operation.



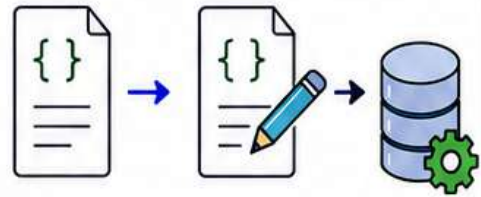
Syntax:

```
db.collection.replaceOne( filter , replacement , options )
```

Example:

// Replace the first document where name is "Bhavitha" with a new document

```
db.students.replaceOne(  
  { name: "Bhavitha" },  
  {  
    _id: ObjectId("663da0c1b8f4a1a5b6c7d8e9"),  
    name: "Bhavitha",  
    age: 23,  
    course: "MTech CSE",  
    city: "Bangalore"  
  }  
)
```



Output:

```
{  
  "acknowledged" : true,  
  "matchedCount" : 1,  
  "modifiedCount" : 1,  
  "upsertedCount" : 0  
}
```

matchedCount - number of documents matched by the filter.
modifiedCount - number of documents actually replaced.
upsertedCount - number of new documents inserted (0 in this case).

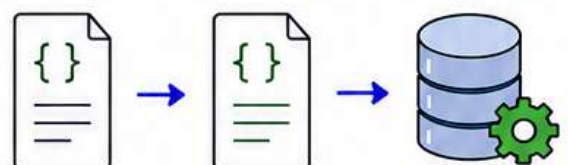
Key Points:

- ✓ `replaceOne()` replaces only **one document** that matches the **filter**.
- ✓ It replaces the **entire document** with the **new document** provided.
- ✓ If no document matches, **matchedCount** will be 0.
- ✓ The replacement document must contain **all required fields**.
- ✓ Use `replaceOne()` when you want to completely replace a document.



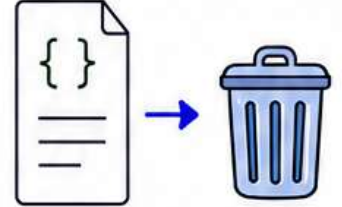
Common Use Case:

- Used when we want to completely replace an existing record with a new one, keeping the same `_id`.



15.8 deleteOne()

- * deleteOne() is used to delete a **single document** from a collection.
- * It deletes the **first document** that matches the filter.
- * If no document matches the filter, **no deletion** is performed.
- * It returns a **result object** with details about the delete operation.
- * It is useful when we want to remove only **one specific document**.

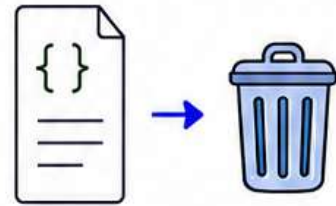


Syntax:

```
db.collection.deleteOne( filter , options )
```

Example:

```
// Delete the first document where name is "Bhavitha"
db.students.deleteOne(
  { name: "Bhavitha" }
)
```



Output:

```
{
  "acknowledged" : true,
  "deletedCount" : 1
}
```

deletedCount - number of documents deleted by the operation.

Key Points:

- ✓ deleteOne() deletes only **one document** that matches the **filter**.
- ✓ It deletes the **first matching** document.
- ✓ If no document matches, **deletedCount** will be **0**.
- ✓ Use it when you want to remove a **single specific record**.
- ✓ The **_id** field is commonly used in the filter for deletion.

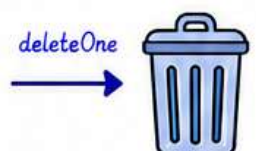


Common Use Case:

- Used when we need to delete a single record, such as removing an inactive user or an obsolete entry.

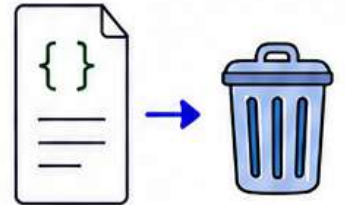
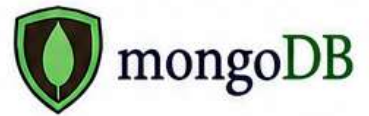
Collection (students)

_id	name	course	age
1	Bhavitha	BTech CSE	22
2	Rohan	BTech ECE	21
3	Sneha	BTech IT	20



15.9 deleteMany()

- * deleteMany() is used to delete **multiple documents** from a collection.
- * It deletes **all documents** that match the filter.
- * If no documents match the filter, **no deletion** is performed.
- * It returns a **result object** with details about the delete operation.
- * It is useful when we want to remove **more than one document** at once.

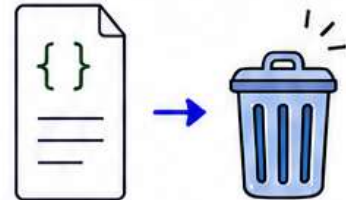


Syntax:

```
db.collection.deleteMany( filter , options )
```

Example:

```
// Delete all documents where course is "BTech CSE"
db.students.deleteMany(
  { course: "BTech CSE" }
)
```



Output:

```
{
  "acknowledged" : true,
  "deletedCount" : 3
}
```

deletedCount – number of documents deleted by the operation.

Key Points:

- ✓ deleteMany() deletes **all documents** that match the **filter**.
- ✓ If no documents match, **deletedCount** will be 0.
- ✓ Use it when you want to remove **multiple records** at once.
- ✓ Be careful while using deleteMany() without a filter (it will delete all documents in the collection).
- ✓ The **_id** field is commonly used in the filter for deletion.



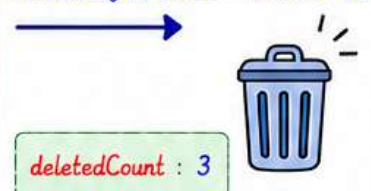
Common Use Case:

- Used when we need to delete multiple records that share a common condition, such as removing inactive users or old log entries.

Collection (students)

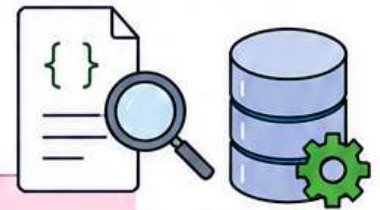
_id	name	course	status
1	Bhavitha	BTech CSE	inactive
2	Rohan	BTech CSE	inactive
3	Sneha	BTech IT	active
4	Arjun	BTech ECE	inactive

deleteMany({ status: "inactive" })



15.10 Query Operators

- * Query operators are used to **filter documents** in a collection.
- * They are used with methods like `find()`, `updateMany()`, `deleteMany()`, etc.
- * Operators help to **compare values**, **check conditions**, or **match patterns**.
- * MongoDB provides a rich set of query operators.
- * We use them inside the **filter** part of the query.



Common Query Operators

Category	Operator	Description	Example (field: age)
Comparison Operators	<code>\$eq</code>	Equals	<code>{ age: { \$eq: 20 } }</code>
	<code>\$ne</code>	Not equals	<code>{ age: { \$ne: 20 } }</code>
	<code>\$gt</code>	Greater than	<code>{ age: { \$gt: 20 } }</code>
	<code>\$gte</code>	Greater than or equal to	<code>{ age: { \$gte: 20 } }</code>
	<code>\$lt</code>	Less than	<code>{ age: { \$lt: 20 } }</code>
	<code>\$lte</code>	Less than or equal to	<code>{ age: { \$lte: 20 } }</code>
	<code>\$in</code>	In (matches any value in array)	<code>{ age: { \$in: [20, 22, 24] } }</code>
	<code>\$nin</code>	Not in (does not match any in array)	<code>{ age: { \$nin: [20, 22, 24] } }</code>
Logical Operators	<code>\$and</code>	Logical AND (all conditions true)	<code>{ \$and: [{ age: { \$gt: 20 } }, { city: "Hyd" }] }</code>
	<code>\$or</code>	Logical OR (any condition true)	<code>{ \$or: [{ age: { \$lt: 20 } }, { city: "Hyd" }] }</code>
	<code>\$not</code>	Logical NOT (negates condition)	<code>{ age: { \$not: { \$gt: 20 } } }</code>
Element Operators	<code>\$exists</code>	Checks if field exists	<code>{ age: { \$exists: true } }</code>
	<code>\$type</code>	Matches BSON type of field	<code>{ age: { \$type: "int" } }</code>
Evaluation Operators	<code>\$regex</code>	Matches using regular expression	<code>{ name: { \$regex: "^Bha" } }</code>
	<code>\$where</code>	Matches using JavaScript expression	<code>{ \$where: "this.age > 20" }</code>

Examples

```
// Find students with age greater than 20
db.students.find({ age: { $gt: 20 } })
```

```
// Find students whose name starts with "B"
db.students.find({ name: { $regex: "^B" } })
```

```
// Find students with course either "CSE" or "ECE"
db.students.find(
  { $or: [ { course: "CSE" }, { course: "ECE" } ] })
```

```
// Find students where city field exists
db.students.find({ city: { $exists: true } })
```

Key Points:

- ✓ Query operators are used to **filter documents** based on conditions.
- ✓ They are used inside the **filter** object of `find()`, `updateMany()`, `deleteMany()`, etc.
- ✓ **Comparison**, **logical**, **element**, and **evaluation** operators help in building powerful queries.
- ✓ Operators can be combined to create complex queries.



Quick Reference (Comparison Operators)

Operator	Meaning	Example (age)
<code>\$eq</code>	equals	<code>{ age: { \$eq: 20 } }</code>
<code>\$ne</code>	not equals	<code>{ age: { \$ne: 20 } }</code>
<code>\$gt</code>	greater than	<code>{ age: { \$gt: 20 } }</code>
<code>\$gte</code>	greater than or equal	<code>{ age: { \$gte: 20 } }</code>
<code>\$lt</code>	less than	<code>{ age: { \$lt: 20 } }</code>
<code>\$lte</code>	less than or equal	<code>{ age: { \$lte: 20 } }</code>

Tip:

- * Use `$in` instead of multiple `$or` conditions for the same field.

Example:

```
db.students.find({ age: { $in: [20, 21, 22] } })
```



15.11 Comparison Operators

- * Comparison operators are used to **compare values** in a query.
- * These operators are mainly used with **numeric, date, or string** values.
- * They are used in the filter part of methods like `find()`, `updateMany()`, `deleteMany()`, etc.
- * They help to **match documents** based on conditions.



Comparison Operators in MongoDB

Operator	Meaning	Syntax (Example)	Description
<code>\$eq</code>	equals	<code>{ age: { \$eq: 20 } }</code>	Selects documents where field equals the value.
<code>\$ne</code>	not equals	<code>{ age: { \$ne: 20 } }</code>	Selects documents where field does not equal the value.
<code>\$gt</code>	greater than	<code>{ age: { \$gt: 20 } }</code>	Selects documents where field is greater than the value.
<code>\$gte</code>	greater than or equal to	<code>{ age: { \$gte: 20 } }</code>	Selects documents where field is greater than or equal to the value.
<code>\$lt</code>	less than	<code>{ age: { \$lt: 20 } }</code>	Selects documents where field is less than the value.
<code>\$lte</code>	less than or equal to	<code>{ age: { \$lte: 20 } }</code>	Selects documents where field is less than or equal to the value.
<code>\$in</code>	in (matches any value in array)	<code>{ age: { \$in: [20, 22, 24] } }</code>	Selects documents where field matches any value in the array.
<code>\$nin</code>	not in (does not match any value in array)	<code>{ age: { \$nin: [20, 22, 24] } }</code>	Selects documents where field does not match any value in array.

Examples (Collection: `students`)

```
// Find students with age greater than 20
db.students.find({ age: { $gt: 20 } })
```

Output:

```
{ _id: 2, name: "Rohan", age: 21 }
{ _id: 3, name: "Sneha", age: 22 }
{ _id: 4, name: "Arjun", age: 24 }
```



```
// Find students with age less than or equal to 20
db.students.find({ age: { $lte: 20 } })
```

Output:

```
{ _id: 1, name: "Bhavitha", age: 20 }
```



```
// Find students with age in the list [20, 22]
db.students.find({ age: { $in: [20, 22] } })
```

Output:

```
{ _id: 1, name: "Bhavitha", age: 20 }
{ _id: 3, name: "Sneha", age: 22 }
```



```
// Find students with age not in the list [20, 22]
db.students.find({ age: { $nin: [20, 22] } })
```

Output:

```
{ _id: 2, name: "Rohan", age: 21 }
{ _id: 4, name: "Arjun", age: 24 }
```



Key Points:

- ✓ Comparison operators **compare field values** with given values.
- ✓ They are used inside the **filter** object of queries.
- ✓ Useful for **numeric, date, and string** comparisons.
- ✓ **`$in`** and **`$nin`** work with arrays of values.
- ✓ These operators can be used with other logical operators like **`$and`**, **`$or`**, etc.



Compare Values

Tip:

- * Use comparison operators to filter documents based on conditions like age, price, date, marks, etc.



15.12 Logical Operators

- * Logical operators are used to combine **multiple conditions** in a query.
- * They are used in the filter part of methods like `find()`, `updateMany()`, `deleteMany()`, etc.
- * They evaluate expressions and return **true** or **false**.
- * MongoDB provides three logical operators: **\$and**, **\$or**, **\$not**.
- * These operators help to build complex queries.



Logical Operators in MongoDB

Operator	Meaning	Syntax (Example)	Description
\$and	Logical AND (all conditions must be true)	<code>{ \$and: [{ condition1 }, { condition2 }, ...] }</code>	Selects documents that satisfy all the specified conditions.
\$or	Logical OR (any one condition must be true)	<code>{ \$or: [{ condition1 }, { condition2 }, ...] }</code>	Selects documents that satisfy at least one of the conditions.
\$not	Logical NOT (negates the condition)	<code>{ field: { \$not: { condition } } }</code>	Selects documents that do not satisfy the condition.

Examples (Collection: `students`)

1. Using **\$and** (All conditions must be true)

```
db.students.find({ $and: [
  { age: { $gte: 20 } },
  { city: "Hyd" }
]})
```

Output:

```
{ _id: 2, name: "Rohan", age: 21, city: "Hyd" }
{ _id: 5, name: "Arjun", age: 24, city: "Hyd" }
```

2. Using **\$or** (Any one condition must be true)

```
db.students.find({ $or: [
  { course: "CSE" },
  { city: "Delhi" }
]})
```

Output:

```
{ _id: 1, name: "Bhavitha", course: "CSE", city: "Hyd" }
{ _id: 3, name: "Sneha", course: "ECE", city: "Delhi" }
{ _id: 4, name: "Kiran", course: "CSE", city: "Pune" }
```

3. Using **\$not** (Negates the condition)

```
db.students.find({
  age: { $not: { $lt: 20 } }
})
```

Output:

```
{ _id: 1, name: "Bhavitha", age: 20 }
{ _id: 2, name: "Rohan", age: 21 }
{ _id: 5, name: "Arjun", age: 24 }
```

Combined Example (Using multiple logical operators)

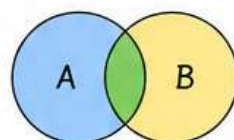
```
db.students.find({ $and: [ { $or: [ { city: "Hyd" }, { city: "Pune" } ] }, { age: { $gte: 20 } } ] })
```

Output:

```
{ _id: 1, name: "Bhavitha", age: 20, city: "Hyd" } { _id: 2, name: "Rohan", age: 21, city: "Hyd" }
```

Key Points:

- ✓ **\$and** returns documents that match **all** the conditions.
- ✓ **\$or** returns documents that match at least **one** condition.
- ✓ **\$not** returns documents that do **not** match the condition.
- ✓ Logical operators are used inside the **filter** object.
- ✓ They help to build **complex** and **powerful** queries.
- ✓ You can combine **\$and**, **\$or** and **\$not** to create advanced query conditions.



\$and → $A \cap B$

\$or → $A \cup B$

\$not A → $\text{not } A$

Tip:

- * Logical operators are very useful when your query contains more than one condition.



15.13 Array Queries

- * Array queries are used to search documents based on **values in an array field**.
- * MongoDB provides special query operators for arrays.
- * These operators are used inside the filter part of methods like `find()`, `updateMany()`, `deleteMany()`, etc.
- * They help to **match elements**, check size, or apply conditions on arrays.



Array Query Operators in MongoDB

Operator	Meaning	Syntax (Example)	Description
\$all	Matches arrays that contain all the specified elements	<code>{ field: { \$all: [value1, value2, ...] } }</code>	Selects documents where the array field contains all the given values.
\$in	Matches any element in the array	<code>{ field: { \$in: [value1, value2, ...] } }</code>	Selects documents where the array field contains at least one of the values.
\$nin	Matches no elements in the array	<code>{ field: { \$nin: [value1, value2, ...] } }</code>	Selects documents where the array field does not contain any of the values.
\$size	Matches arrays with the exact length	<code>{ field: { \$size: number } }</code>	Selects documents where the array field has the exact number of elements.
\$elemMatch	Matches documents that satisfy all the conditions for at least one array element	<code>{ field: { \$elemMatch: { condition1, condition2, ... } } }</code>	Provides advanced filtering for arrays of documents (or complex values).

Examples (Collection: students)

1. Using \$all <pre>db.students.find({ subjects: { \$all: ["DSA", "DBMS"] } })</pre> Output: <pre>{ _id: 1, name: "Bhavitha", subjects: ["DSA", "DBMS", "OS"] } { _id: 3, name: "Sneha", subjects: ["DBMS", "DSA", "CN"] }</pre>	2. Using \$in <pre>db.students.find({ subjects: { \$in: ["DBMS", "OS"] } })</pre> Output: <pre>{ _id: 1, name: "Bhavitha", subjects: ["DSA", "DBMS", "OS"] } { _id: 2, name: "Rohan", subjects: ["OS", "CN"] } { _id: 3, name: "Sneha", subjects: ["DBMS", "DSA", "CN"] }</pre>	3. Using \$nin <pre>db.students.find({ subjects: { \$nin: ["DSA"] } })</pre> Output: <pre>{ _id: 2, name: "Rohan", subjects: ["OS", "CN"] } { _id: 4, name: "Arjun", subjects: ["CN", "OS"] }</pre>
4. Using \$size <pre>db.students.find({ subjects: { \$size: 2 } })</pre> Output: <pre>{ _id: 2, name: "Rohan", subjects: ["OS", "CN"] } { _id: 4, name: "Arjun", subjects: ["CN", "OS"] }</pre>	5. Using \$elemMatch <pre>db.students.find({ scores: { \$elemMatch: { subject: "DBMS", marks: { \$gt: 80 } } } })</pre> Output: <pre>{ _id: 1, name: "Bhavitha", scores: [{ subject: "DBMS", marks: 85 }, { subject: "DSA", marks: 90 }] } { _id: 3, name: "Sneha", scores: [{ subject: "DBMS", marks: 92 }, { subject: "CN", marks: 78 }] }</pre>	

Key Points:

- ✓ Array queries are used to search based on elements in an array field.
- ✓ **\$all** matches all specified values.
- ✓ **\$in** matches any one of the specified values.
- ✓ **\$nin** matches none of the specified values.
- ✓ **\$size** matches arrays with the exact length.
- ✓ **\$elemMatch** is used for complex conditions on array elements.
- ✓ These operators make array filtering powerful and flexible.



Tip:

- * Array operators are used inside the **filter** object of queries.
- They are very useful when your data contains lists or collections of values.

Sample Collection: students

_id	name	subjects	scores (array of documents)
1	Bhavitha	["DSA", "DBMS", "OS"]	[{ "subject": "DBMS", "marks": 85 }, { "subject": "DSA", "marks": 90 }]
2	Rohan	["OS", "CN"]	[{ "subject": "OS", "marks": 75 }, { "subject": "CN", "marks": 80 }]
3	Sneha	["DBMS", "DSA", "CN"]	[{ "subject": "DBMS", "marks": 92 }, { "subject": "CN", "marks": 78 }]
4	Arjun	["CN", "OS"]	[{ "subject": "CN", "marks": 68 }, { "subject": "OS", "marks": 71 }]



15.14 Embedded Documents

- * Embedded documents are documents stored **inside another document**.
- * They are used to represent relationships (**one-to-one** or **one-to-few**).
- * Embedding **reduces the need for joins** and improves read performance.
- * MongoDB allows documents to have **nested documents** or **arrays of documents**.
- * They are stored in BSON format like any other field.



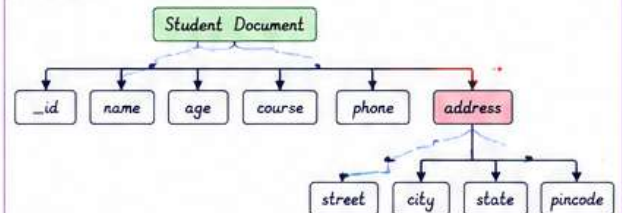
Example of Embedded Document

```
// A student document with an embedded address document
{
  "_id"      : 1,
  "name"     : "Bhavitha",
  "age"      : 21,
  "course"   : "BTech CSE",
  "address"  : {
    "street"  : "H.No: 12-5-66",
    "city"    : "Hyderabad",
    "state"   : "Telangana",
    "pincode" : 500081
  },
  "phone"    : "9876543210"
}
```

Explanation:

- * The **"address"** field is an embedded document **inside** the student document.
- * All fields of **address** are stored as part of the same document.
- * This is useful when the embedded data is always accessed along with the parent document.

Structure



Querying Embedded Documents

We can query fields inside embedded documents using **dot notation**.

1. Find students from city "Hyderabad"

```
db.students.find( { "address.city": "Hyderabad" } )
```

Output:

```
{ _id: 1, name: "Bhavitha", city: "Hyderabad" }
```

2. Find students with pincode 500081

```
db.students.find( { "address.pincode": 500081 } )
```

Output:

```
{ _id: 1, name: "Bhavitha", pincode: 500081 }
```

3. Find students with state "Telangana"

```
db.students.find( { "address.state": "Telangana" } )
```

Output:

```
{ _id: 1, name: "Bhavitha", state: "Telangana" }
```

Arrays of Embedded Documents

We can also store an **array of embedded documents**.

```
// A student with multiple marks (array of documents)
{
  "_id"      : 2,
  "name"     : "Rohan",
  "marks"    : [
    { "subject": "DSA",      "marks": 85 },
    { "subject": "DBMS",    "marks": 92 },
    { "subject": "OS",      "marks": 78 }
  ]
}
```

Query: Find students who scored more than 80 in any subject

```
db.students.find( { "marks.marks": { $gt: 80 } } )
```

Output:

```
{ _id: 2, name: "Rohan" }
```

Explanation:

"marks" is an array of embedded documents.
We query using dot notation inside the array.



Key Points:

- ✓ Embedded documents are stored inside another document.
- ✓ They are used for one-to-one or **one-to-few** relationships.
- ✓ They improve read performance and reduce the need for joins.
- ✓ Use **dot notation** to query fields inside embedded documents.
- ✓ Arrays can contain embedded documents.
- ✓ Use embedding when the data is closely related and accessed together.



When to Use Embedding?

Use Embedding When	Use References When
Data is tightly related.	Data is large and used in many places.
You always access the data together.	You need to update independently.
One-to-one or one-to-few.	One-to-many or many-to-many.
You want better read performance.	You want to avoid data duplication.

Tip:

- * Embedding is great for small, related data. But if the embedded document grows too large, consider using references.

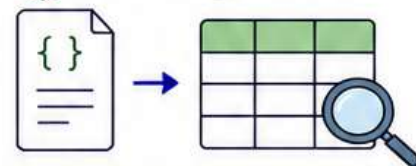


15.15 Projection

- * Projection is used to specify which fields to **include** or **exclude** in the result.
- * It helps to **limit the amount of data** returned by a query.
- * Projection is the second parameter in methods like `find()`.
- * It improves **performance** by sending only the required data.



mongoDB



Syntax

```
db.collection.find( <query>, <projection> )
```

Where, **<query>** : (Optional) specifies the search condition.

<projection> : specifies the fields to include or exclude.

Projection Rules

- * By default, all fields are returned.
- * Use **1** (or **true**) to include a field.
- * Use **0** (or **false**) to exclude a field.
- * You cannot mix inclusion (1) and exclusion (0) in the same projection (except for **_id**).
- * By default, **_id** field is included in the result. To exclude it, set **_id: 0**.

Projection Examples

Type	Description	Projection	Example	Result (Fields Returned)
Inclusion	Include only specified fields.	{ field1: 1, field2: 1 }	db.students.find({}, { name: 1, age: 1 })	Returns only name and age (fields _id , name , age)
Exclusion	Exclude the specified fields.	{ field1: 0, field2: 0 }	db.students.find({}, { marks: 0, address: 0 })	Returns all fields except marks and address
Exclude _id	Exclude only the _id field.	{ _id: 0 }	db.students.find({}, { _id: 0 })	Returns all fields except _id
Include Specific & Exclude _id	Include certain fields and exclude _id .	{ field1: 1, field2: 1, _id: 0 }	db.students.find({}, { name: 1, age: 1, _id: 0 })	Returns only name and age without _id

Sample Collection: students

```
{ _id: 1, name: "Bhavitha", age: 21,
  course: "CSE", city: "Hyderabad", marks: 85 }
{ _id: 2, name: "Rohan", age: 22,
  course: "ECE", city: "Delhi", marks: 78 }
{ _id: 3, name: "Sneha", age: 21,
  course: "IT", city: "Pune", marks: 92 }
```

Note:

If you do not specify a projection, MongoDB returns all fields of the matching documents.



More Examples

1. Include name and course

```
db.students.find( {}, { name: 1, course: 1 } )
```

Output:

```
{ _id: 1, name: "Bhavitha", course: "CSE" }
{ _id: 2, name: "Rohan", course: "ECE" }
{ _id: 3, name: "Sneha", course: "IT" }
```



2. Exclude marks

```
db.students.find( {}, { marks: 0 } )
```

Output:

```
{ _id: 1, name: "Bhavitha", age: 21, course: "CSE", city: "Hyderabad" }
{ _id: 2, name: "Rohan", age: 22, course: "ECE", city: "Delhi" }
{ _id: 3, name: "Sneha", age: 21, course: "IT", city: "Pune" }
```



3. Exclude city and **_id**

```
db.students.find( {}, { city: 0, _id: 0 } )
```

Output:

```
{ name: "Bhavitha", age: 21, course: "CSE", marks: 85 }
{ name: "Rohan", age: 22, course: "ECE", marks: 78 }
{ name: "Sneha", age: 21, course: "IT", marks: 92 }
```



4. Include name and city, exclude **_id**

```
db.students.find( {}, { name: 1, city: 1, _id: 0 } )
```

Output:

```
{ name: "Bhavitha", city: "Hyderabad" }
{ name: "Rohan", city: "Delhi" }
{ name: "Sneha", city: "Pune" }
```



Key Points:

- ✓ Projection is used to control which fields are returned.
- ✓ Use **1** to include fields, **0** to exclude fields.
- ✓ Do not mix inclusion and exclusion in the same projection.
- ✓ Use **_id: 0** to exclude the **_id** field.



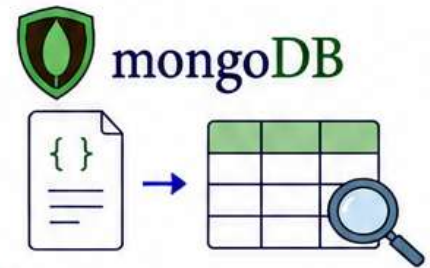
Tip:

- * Use projection to improve performance and reduce network traffic by returning only the necessary data.



15.16 Sorting

- * Sorting is used to **arrange** the result documents in a **specific order**.
- * MongoDB provides the **sort()** method to sort the results.
- * The **sort()** method accepts **field(s)** and **order** as parameters.
- * Order can be **1** for **ascending** or **-1** for **descending**.
- * Default order is **ascending (1)**.



Syntax

```
db.collection.find( <query>, <projection> ).sort( { field: order, ... } )
```

Where,

- <query>** : (Optional) specifies the search condition.
- <projection>** : (Optional) specifies the fields to return.
- field** : field name to sort by.
- order** : **1** for ascending, **-1** for descending.

Sorting Rules

- * **1** → **Ascending** order (A to Z, 0 to 9).
- * **-1** → **Descending** order (Z to A, 9 to 0).
- * You can sort by **one field** or **multiple fields**.
- * Sort is applied after filtering (**where**) and before limiting (**limit**).
- * If no **sort()** is used, order is **not guaranteed**.

Sort Order Table

Order	Value	Description	Example
Ascending	1	Sorts the result in increasing order. (A → Z, 0 → 9)	<code>db.students.find().sort({ name: 1 })</code>
Descending	-1	Sorts the result in decreasing order. (Z → A, 9 → 0)	<code>db.students.find().sort({ name: -1 })</code>

Sample Collection: students

```
{ _id: 1, name: "Bhavitha", age: 21, city: "Hyderabad" }
{ _id: 2, name: "Rohan", age: 22, city: "Delhi" }
{ _id: 3, name: "Sneha", age: 21, city: "Pune" }
{ _id: 4, name: "Arjun", age: 23, city: "Chennai" }
{ _id: 5, name: "Kiran", age: 22, city: "Hyderabad" }
```

Notes:

- * Sorting is case sensitive for strings.
- * When two documents have the same value for the sort field, MongoDB may return them in any order unless you sort by additional fields.
- * You can sort on fields that are included in the result (projection) or not.

Sorting Examples

1. Sort by name (Ascending)

```
db.students.find().sort( { name: 1 } )
```

Output:

```
{ _id: 4, name: "Arjun", age: 23, city: "Chennai" }
{ _id: 1, name: "Bhavitha", age: 21, city: "Hyderabad" }
{ _id: 5, name: "Kiran", age: 22, city: "Hyderabad" }
{ _id: 2, name: "Rohan", age: 22, city: "Delhi" }
{ _id: 3, name: "Sneha", age: 21, city: "Pune" }
```

2. Sort by age (Descending)

```
db.students.find().sort( { age: -1 } )
```

Output:

```
{ _id: 4, name: "Arjun", age: 23, city: "Chennai" }
{ _id: 2, name: "Rohan", age: 22, city: "Delhi" }
{ _id: 5, name: "Kiran", age: 22, city: "Hyderabad" }
{ _id: 1, name: "Bhavitha", age: 21, city: "Hyderabad" }
{ _id: 3, name: "Sneha", age: 21, city: "Pune" }
```

3. Sort by city (Ascending)

```
db.students.find().sort( { city: 1 } )
```

Output:

```
{ _id: 2, name: "Rohan", age: 22, city: "Delhi" }
{ _id: 4, name: "Arjun", age: 23, city: "Chennai" }
{ _id: 3, name: "Sneha", age: 21, city: "Pune" }
{ _id: 1, name: "Bhavitha", age: 21, city: "Hyderabad" }
{ _id: 5, name: "Kiran", age: 22, city: "Hyderabad" }
```

4. Sort by age (Ascending) and name (Ascending)

```
db.students.find().sort( { age: 1, name: 1 } )
```

Output:

```
{ _id: 1, name: "Bhavitha", age: 21, city: "Hyderabad" }
{ _id: 3, name: "Sneha", age: 21, city: "Pune" }
{ _id: 2, name: "Rohan", age: 22, city: "Delhi" }
{ _id: 5, name: "Kiran", age: 22, city: "Hyderabad" }
{ _id: 4, name: "Arjun", age: 23, city: "Chennai" }
```

Key Points:

- ✓ Use **sort()** to arrange documents in a specific order.
- ✓ **1** → Ascending order (default), **-1** → Descending order.
- ✓ You can sort by one field or multiple fields.
- ✓ Sorting improves the readability of results.
- ✓ Useful with **limit()** to get top N records.



Tip:

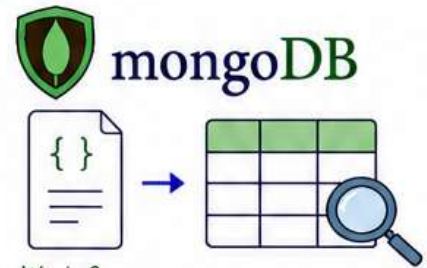
- * Use **sort()** along with **limit()** to get top results.

```
db.students.find().sort( { marks: -1 } ).limit( 3 )
```



15.17 Pagination

- * Pagination is used to **divide the result** into smaller pages.
- * MongoDB provides **skip()** and **limit()** methods for pagination.
- * **skip()** is used to skip a number of documents.
- * **limit()** is used to specify the number of documents to return.
- * Pagination is useful for **large datasets** and improves performance.



Syntax

```
db.collection.find( <query>, <projection> )
    .sort( { <field>: <order> } )    // (Optional)
    .skip( <number> )                // Number of documents to skip
    .limit( <number> )              // Number of documents to return
```

Where,

<query> : (Optional) specifies the search condition.
<projection> : (Optional) specifies the fields to return.
<field> : field name to sort by (recommended for pagination).
<order> : 1 for ascending, -1 for descending.
skip(<number>) : skips the first <number> documents.
limit(<number>) : limits the result to <number> documents.

How Pagination Works?

- * Page number starts from 1.
- * Page size (limit) is the number of documents in each page.
- * To get page N with page size P:

$$\begin{aligned}\text{skip} &= (N - 1) \times P \\ \text{limit} &= P\end{aligned}$$

- * Always use **sort()** with pagination to get consistent results.
- * Without **sort()**, results may change between pages.

Sample Collection: students

```
{ _id: 1, name: "Bhavitha", age: 21, city: "Hyderabad" }
{ _id: 2, name: "Rohan", age: 22, city: "Delhi" }
{ _id: 3, name: "Sneha", age: 21, city: "Pune" }
{ _id: 4, name: "Arjun", age: 23, city: "Chennai" }
{ _id: 5, name: "Kiran", age: 22, city: "Hyderabad" }
{ _id: 6, name: "Megha", age: 20, city: "Delhi" }
{ _id: 7, name: "Vikram", age: 24, city: "Pune" }
{ _id: 8, name: "Anjali", age: 21, city: "Chennai" }
{ _id: 9, name: "Sagar", age: 22, city: "Hyderabad" }
{ _id: 10, name: "Neha", age: 23, city: "Delhi" }
```

(Total Documents = 10)

Pagination Summary (Page Size = 3)

Page No.	skip()	limit()	Documents Returned (sorted by _id)
1	0	3	_id: 1, 2, 3
2	3	3	_id: 4, 5, 6
3	6	3	_id: 7, 8, 9
4	9	3	_id: 10

Formula: $\text{skip} = (\text{Page No.} - 1) \times \text{Page Size}$
 $\text{limit} = \text{Page Size}$

Pagination Examples (Page Size = 3, sorted by _id)

1. Get Page 1

```
db.students.find({}).sort({_id: 1})
    .skip( 0 ).limit( 3 )
```

Output:

```
{ _id: 1, name: "Bhavitha" }
{ _id: 2, name: "Rohan" }
{ _id: 3, name: "Sneha" }
```

2. Get Page 2

```
db.students.find({}).sort({_id: 1})
    .skip( 3 ).limit( 3 )
```

Output:

```
{ _id: 4, name: "Arjun" }
{ _id: 5, name: "Kiran" }
{ _id: 6, name: "Megha" }
```

3. Get Page 3

```
db.students.find({}).sort({_id: 1})
    .skip( 6 ).limit( 3 )
```

Output:

```
{ _id: 7, name: "Vikram" }
{ _id: 8, name: "Anjali" }
{ _id: 9, name: "Sagar" }
```

4. Get Page 4

```
db.students.find({}).sort({_id: 1})
    .skip( 9 ).limit( 3 )
```

Output:

```
{ _id: 10, name: "Neha" }
```

Key Points:

- ✓ Use **skip()** to skip documents.
- ✓ Use **limit()** to restrict the number of documents.
- ✓ Always use **sort()** for consistent and reliable pagination.
- ✓ Choose a proper **page size** (not too small, not too large).
- ✓ Avoid using very large **skip()** values for big datasets as it can affect performance.



Get Total Number of Documents

```
db.students.countDocuments( <query> )
```

Example:

```
db.students.countDocuments( {} )
```

Output: 10



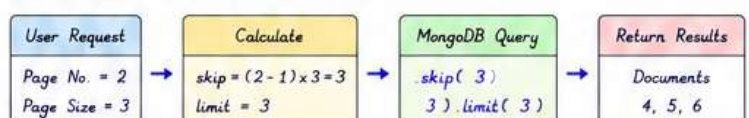
Tip:

- * For better performance on large collections, create an index on the field used in **sort()**.

Example: `db.students.createIndex( { _id: 1 } )`

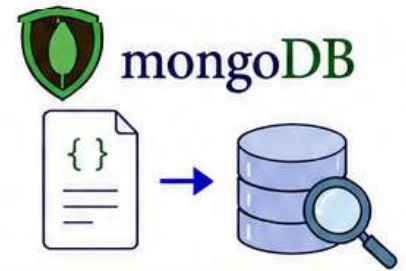


Typical Pagination Flow in Application



15.18 MongoDB Indexes

- * Indexes **improve** the **speed of read operations** on a collection.
- * MongoDB indexes store a small portion of the data in an easy to traverse form.
- * Without indexes, MongoDB must **scan the entire collection**.
- * Indexes consume additional disk space.
- * MongoDB creates an **_id** index by **default** on every collection.



Key Points

- * **1** → Ascending index (A to Z, 0 to 9).
- * **-1** → Descending index (Z to A, 9 to 0).
- * Indexes can be created on single field or multiple fields (compound index).
- * Indexes **improve performance** of queries that use the indexed fields in filter or sort.
- * Too many indexes may **slow down write** operations.

Syntax

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

Where,

- field** : Field name to create index on.
- 1** : Ascending order index.
- 1** : Descending order index.

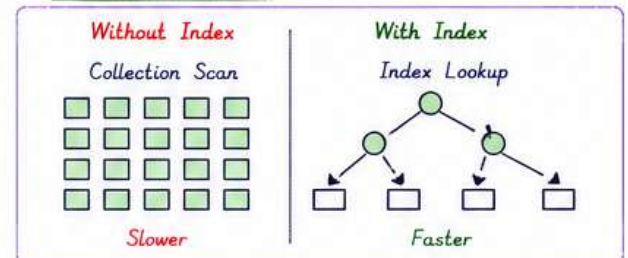
Types of Indexes

Index Type	Description	createIndex() Example
Single Field Index	Index on a single field.	db.students.createIndex({ name: 1 })
Compound Index	Index on multiple fields.	db.students.createIndex({ age: 1, city: -1 })
Unique Index	Ensures all values in the indexed field are unique.	db.students.createIndex({ email: 1 }, { unique: true })
Sparse Index	Indexes only documents that contain the field.	db.students.createIndex({ middleName: 1 }, { sparse: true })
TTL Index	Automatically removes documents after a specified time.	db.sessions.createIndex({ lastAccess: 1 }, { expireAfterSeconds: 3600 })
Text Index	Supports text search.	db.students.createIndex({ bio: "text" })
Hashed Index	Indexes the hash value of a field.	db.users.createIndex({ userId: "hashed" })

Example Collection: students

```
{ _id: 1, name: "Bhavitha", age: 21, city: "Hyderabad", email: "b@abc.com" }
{ _id: 2, name: "Rohan", age: 22, city: "Delhi", email: "r@abc.com" }
{ _id: 3, name: "Sneha", age: 21, city: "Pune", email: "s@abc.com" }
{ _id: 4, name: "Arjun", age: 23, city: "Chennai", email: "a@abc.com" }
```

How Indexes Work?



Create Index Examples

1. Single Field Index (Ascending)
db.students.createIndex({ name: 1 })
2. Single Field Index (Descending)
db.students.createIndex({ age: -1 })
3. Compound Index
db.students.createIndex({ age: 1, city: -1 })
4. Unique Index
db.students.createIndex({ email: 1 }, { unique: true })
5. Sparse Index
db.students.createIndex({ middleName: 1 }, { sparse: true })
6. TTL Index (expire after 1 hour)
db.sessions.createIndex({ lastAccess: 1 }, { expireAfterSeconds: 3600 })
7. Text Index
db.students.createIndex({ bio: "text" })
8. Hashed Index
db.users.createIndex({ userId: "hashed" })

View Existing Indexes

```
db.collection.getIndexes( )
```

Example:

```
db.students.getIndexes( )
```

Output:

```
[ { "v": 2, "key": { "_id": 1 }, "name": "_id_",
  { "v": 2, "key": { "name": 1 }, "name": "name_1" } ]
```

Drop (Remove) an Index

```
db.collection.dropIndex( "index_name" )
```

Example:

```
db.students.dropIndex( "name_1" ) // drops index on name field
```

Best Practices

- ✓ Index fields that are frequently queried.
- ✓ Index fields used in sort() operations.
- ✓ Use compound indexes for queries on multiple fields.
- ✓ Avoid indexing fields with very few distinct values.
- ✓ Regularly review and remove unused indexes.



Important Notes

- * Choose indexes based on fields used in queries, sorting and filtering.
- * Create only necessary indexes.
- * Monitor index usage using explain().
- * Too many indexes increase storage and slow down writes.



Tip

- * Indexing is powerful, but over-indexing can hurt performance. Always test and choose the right indexes for your queries!

