Class 1 : Introduction to MongoDB

MongoDB Installation Download and Install MongoDB on windows using

https://www.mongodb.com/download-center/community Download MongoDB shell using https://fastdl.mongodb.org/windows/mongodb-windows-x86_64-7.0.11-signed.msi

MongoDB is an open-source document-oriented database that is designed to store a large scale of data and also allows you to work with that data very efficiently. It is categorized under the NoSQL database because the storage and retrieval of data in the MongoDB are not in the form of tables.

- **Structured Data:**

The information is typically organized in a specific format, often using tables with rows and columns. This makes it easier to search, filter, and analyze the data.

- **Database Management System (DBMS):**

This is the software that acts like the filing cabinet manager. It allows you to store, retrieve, update, and manage all the data within the database.

- **Data Types:**

Databases can hold various kinds of information, including text, numbers, images, videos, and more.

- **MongoDB vs. MySQL**

MySQL (link resides outside IBM.com) uses a structured query language to access stored data. In this format, schemas are used to create database structures, utilizing tables as a way to standardize data types so that values are searchable and can be queried properly. A mature solution, MySQL is useful for a variety of situations including website databases, applications and commercial product management.

- **Advantage :**

Schema Flexibility:

- MongoDB stores data in flexible, JSON-like documents. This allows for the storage of semi-structured data and the ability to change the data structure over time without significant downtime or reengineering.

Scalability:

- MongoDB is designed to scale out horizontally by sharding, distributing data across multiple servers. This makes it easier to manage large volumes of data and handle high-throughput operations.

High Performance:

- With its ability to index and query data efficiently, MongoDB can deliver high performance for both read and write operations. It supports rich queries, indexing, and aggregation.

Geospatial Support:

- MongoDB has built-in support for geospatial data and queries, making it suitable for applications that require location-based features.

Aggregation Framework:

- MongoDB provides a powerful aggregation framework that allows for complex data processing and transformation tasks within the database, reducing the need for additional processing in the application layer.

Document-Based Model:

- The document model maps naturally to objects in application code, making it easy for developers to work with data. This is particularly beneficial for object-oriented programming.

Open Source:

MongoDB is open-source, which means it is free to use and has a large, active community contributing to its development and providing support.

Strong Ecosystem:

- MongoDB has a strong ecosystem with a wide range of tools, libraries, and integrations. This includes support for various programming languages, data visualization tools, and cloud services.

Disadvantage :

Data Size and Memory Usage:

- MongoDB can consume a significant amount of memory and storage due to its schema-less design and the storage format of its BSON documents, which might include redundant data and larger document sizes.

Limited Support for Complex Transactions:

- Although MongoDB supports multi-document ACID transactions starting from version 4.0, its transaction support is still not as robust as traditional relational databases. Complex transactions involving many documents can be less efficient and more challenging to manage.

Consistency Concerns:

By default, MongoDB is configured for eventual consistency, which might not be suitable for applications requiring strong consistency. This means that there can be a delay before all nodes in a distributed system reflect a write operation. Indexing Limitations:

- While MongoDB supports indexing, there are limitations on the number of indexes per collection and the size of indexed fields. Poor indexing strategies can lead to significant performance issues.

Join Limitations:

- MongoDB does not support joins in the traditional sense found in relational databases. While it has the $lookup operation to perform joins, these operations can be less efficient and more cumbersome for complex queries.

Learning Curve:

- For teams accustomed to relational databases, transitioning to MongoDB can involve a steep learning curve. Developers need to understand NoSQL principles and best practices to effectively use MongoDB.

Data Duplication:

The lack of normalization can lead to data duplication, which might result in increased storage requirements and challenges with data consistency and updates.

Applications

Web Application

Big Data

Demographic and Biometric Data

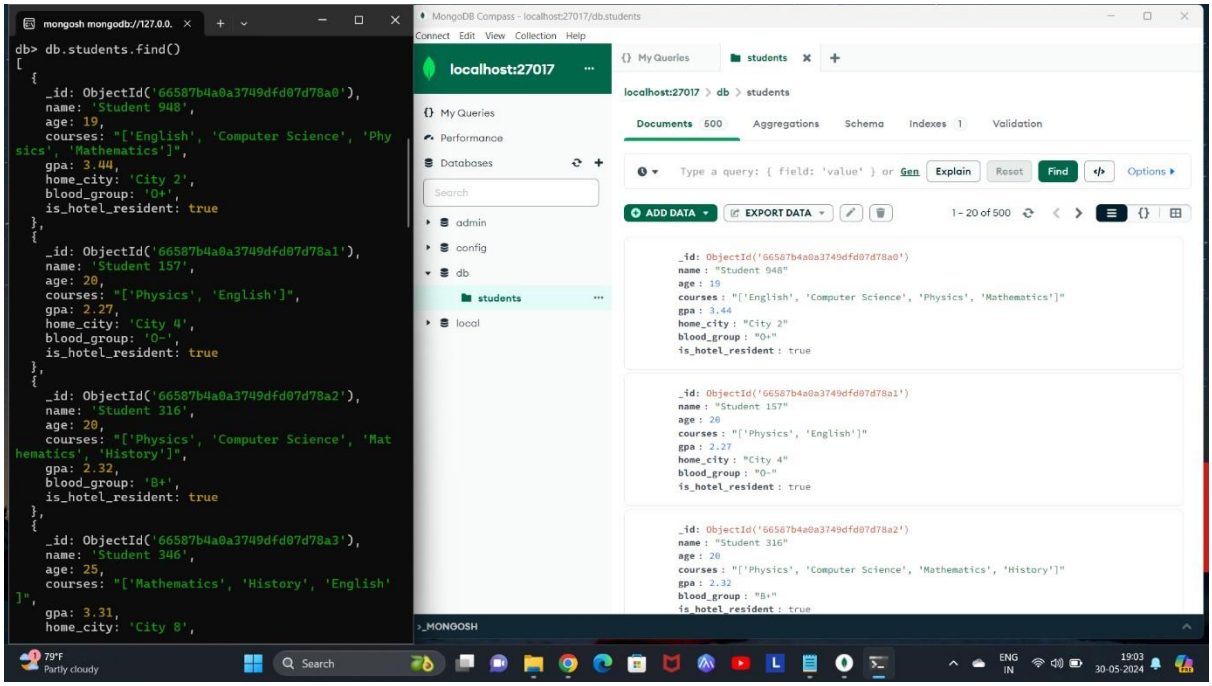
Synchronizatio

Class 2 : Add ,Update and Delete

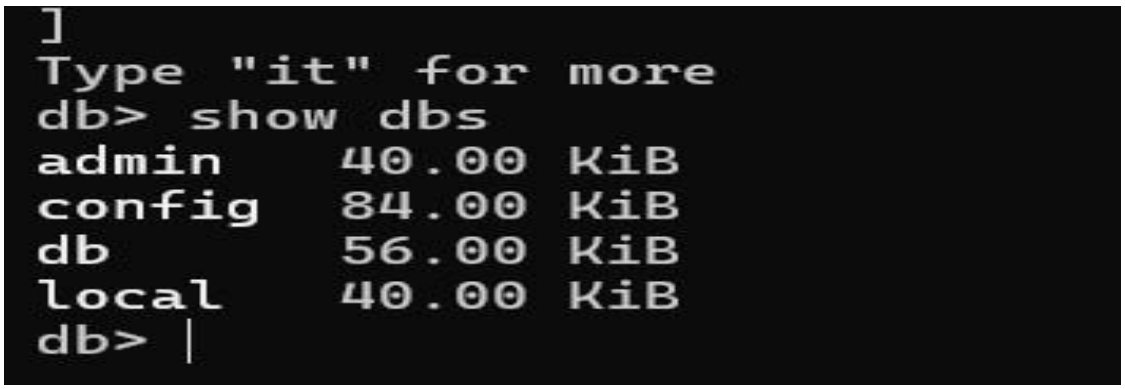
To find the data present in the collections ,we can use the command “db.collection_name.find()”

.

In this the collection name is “students”.



“show dbs” command shows all the database.



- Collections :

A collection is a group of documents.

If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table.

- Database:

MongoDB groups collections into databases.

A single instance of MongoDB can host several databases, each grouping together zero or more collections.

- Document:

At the heart of MongoDB is the document: an ordered set of
keys with associated values.
The representation of a document varies by programming language, but most languages have a data structure that is a natural fit, such as a map, hash, or dictionary.

Class 3 : WHERE, AND, OR & CRUD

- **WHERE**:

WHERE is used when we want to filter a subset based on a condition.

To find the collection with gpa greater than 3.0 we use a command **db.students.find({gpa:{$gt:3.0}});**

```
db> db.students.find({ gpa:{$gt:3}})
[
  {
    _id: ObjectId('66587b4a0a3749dfd07d78a0'),
    name: 'Student 948',
    age: 19,
    courses: "['English', 'Computer Science', 'Phy
sics', 'Mathematics']",
    gpa: 3.44,
    home_city: 'City 2',
    blood_group: 'O+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66587b4a0a3749dfd07d78a3'),
    name: 'Student 346',
    age: 25,
    courses: "['Mathematics', 'History', 'English'
]",
    gpa: 3.31,
    home_city: 'City 8',
    blood_group: 'O-',
    is_hotel_resident: true
  },
```

We use the condition gpa greater than 3. Here it shows students gpa greater than 3.The result is shown is based on this condition.

- **AND:**

AND is used when, in a given collection we want to filter a subset based on multiple conditions.

To find students who are lived in city 1 and having a blood group

"O-"we use **db.students.find({$and : [{home_city:"City1"},{blood_group:"O-"}]});**

```
Type "it" for more
db> db.students.find({
... $and:[
... {home_city:"City 1"},
... {blood_group:"O-"}
... ]
... })
[
  {
    _id: ObjectId('66587b4a0a3749dfd07d78c0'),
    name: 'Student 384',
    age: 18,
    courses: "['Mathematics', 'Computer Science']"
,
    gpa: 3.9,
    home_city: 'City 1',
    blood_group: 'O-',
    is_hotel_resident: false
  },
  {
    _id: ObjectId('66587b4a0a3749dfd07d7950'),
    name: 'Student 702',
    age: 22,
    courses: "['History', 'Mathematics', 'English'
]",
    gpa: 3.74,
    home_city: 'City 1',
    blood_group: 'O-',
    is_hotel_resident: false
  },
```

home_city : City1' and 'blood_group : O-' are the conditions on which it is filtered.

- **OR:**

OR is used when , in a given collection we want to filter a subset based on multiple conditions but any one is sufficient .

Here we are checking for students who are hotel resident and scored gpa less than 3.0 we use **db.students.find({$or:[{is_hotel _resident:true},{gpa:{$lt:3.000}}]});**

```
db> db.students.find({$or:[{is_hotel_resident:true},{gpa:{$lt:3.0}}]});
[
  {
    _id: ObjectId('6663dac4f24355f2c2a837e5'),
    name: 'Student 948',
    age: 19,
    courses: "['English', 'Computer Science', 'Physics', 'Mathematics']",
    gpa: 3.44,
    home_city: 'City 2',
    blood_group: 'O+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('6663dac4f24355f2c2a837e6'),
    name: 'Student 157',
    age: 20,
    courses: "['Physics', 'English']",
    gpa: 2.27,
    home_city: 'City 4',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('6663dac4f24355f2c2a837e7'),
    name: 'Student 316',
    age: 20,
    courses: "['Physics', 'Computer Science', 'Mathematics', 'History']",
    gpa: 2.32,
    blood_group: 'B+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('6663dac4f24355f2c2a837e8'),
    name: 'Student 346',
    age: 25,
    courses: "['Mathematics', 'History', 'English']",
    gpa: 3.31,
    home_city: 'City 8',
```

- **CRUD:**

C – Create / Insert

R – Remove

U – Update D – Delete

This is applicable for a collection (table) or a document (row)

- Update:

'$set' command is used to update the data present in the collection.

```
db> db.students.updateOne( { name:"Sam"} , {$set:{
gpa:3} } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
db>
```

- Delete:

It is used to delete the data present in the collection.

```
db> db.students.deleteOne({ name:"Sam" })
{ acknowledged: true, deletedCount: 1 }
db>
```

- Update many:

update some aspects(saved information) of a students

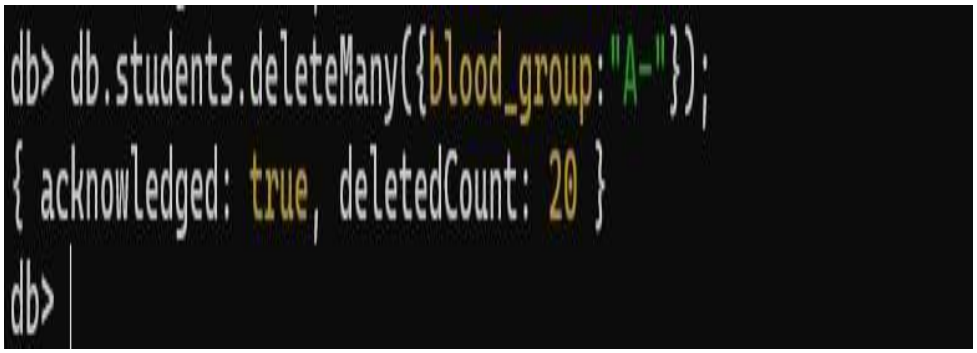
To update data of students with a gpa greater than 3.5 by increasing 0.5 we use **db.students.updateMany({gpa:3.5}},{$inc:{gpa:0.5}});**

```
db> db.students.updateMany({gpa:{$gt:3.5}},{$inc:{gpa:0.5}});
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 124,
  modifiedCount: 124,
  upsertedCount: 0
}
db>
```

- Delete many:

To delete data of students who are hotel_residents we use a command **db.students.deleteMany({is_hotel_resident:true});** and the countings of deleted students is 246.

Delete all the students who's blood group "A-". db.students.deleteMany({blood_group:"A-"});



- Insert:

Here we are inserting the student details name ‘Sam’ and other information to the collection ‘students’.the insertion is done one time.

```
db> const studentData = {
... "name":"Sam",
... "age":22,
... "courses":["Computer Science" , "Mathematics"]
,
... "gpa":3.4,
... "home_city":"City 3",
... "blood_group":"B+",
... "is_hotel_resident":false
... }

db> db.students.insertOne(studentData)
{
  acknowledged: true,
  insertedId: ObjectId('6658a0c70cce0c5ec1cdcdf6')
}
db>
```

Projection:

This is used when we don't need all columns / attributes.

- Benefits of Projection:

  Reduced data transferred between the database and your application.

  Improves query performance by retrieving only necessary data.

  Simplifies your code by focusing on the specific information you need.

```
db> db.students.deleteOne({ name:"Sam" })
{ acknowledged: true, deletedCount: 1 }
db> db.students.find({} , {name:1 , gpa:1 })
[
  {
    _id: ObjectId('66587b4a0a3749dfd07d78a0'),
    name: 'Student 948',
    gpa: 3.44
  },
  {
    _id: ObjectId('66587b4a0a3749dfd07d78a1'),
    name: 'Student 157',
    gpa: 2.27
  },
  {
    _id: ObjectId('66587b4a0a3749dfd07d78a2'),
    name: 'Student 316',
    gpa: 2.32
```

Here it only shows the name and gpa . Because the command is give as 'name:1' and 'gpa:1'.

Class 4 : Projection, Limit & Selectors

Get Selected Attributes:

Given a Collection you want to FILTER a subset of attributes. That is the place Projection is used.

```
db> db.students.find({},{name:1,age:1});
[
  {
    _id: ObjectId('66682c12c4b3310818df6409'),
    name: 'Student 948',
    age: 19
  },
  {
    _id: ObjectId('66682c12c4b3310818df640a'),
    name: 'Student 157',
    age: 20
  },
  {
    _id: ObjectId('66682c12c4b3310818df640b'),
    name: 'Student 316',
    age: 20
  },
  {
    _id: ObjectId('66682c12c4b3310818df640c'),
    name: 'Student 346',
    age: 25
  },
  {
    _id: ObjectId('66682c12c4b3310818df640d'),
    name: 'Student 930',
    age: 25
  },
  {
    _id: ObjectId('66682c12c4b3310818df640e'),
    name: 'Student 305',
    age: 24
  },
  {
    _id: ObjectId('66682c12c4b3310818df640f'),
    name: 'Student 268',
    age: 21
  },
  {
    _id: ObjectId('66682c12c4b3310818df6410'),
    name: 'Student 563',
    age: 18
  },
  {
    _id: ObjectId('66682c12c4b3310818df6411'),
    name: 'Student 440',
    age: 21
  },
  {
    _id: ObjectId('66682c12c4b3310818df6412'),
    name: 'Student 536',
    age: 20
  },
  {
    _id: ObjectId('66682c12c4b3310818df6413'),
    name: 'Student 256',
    age: 19
  },
```

Here we are displaying only name and age. We use the command **db.students.find({},{name:1, age:1});**

Ignore Attributes :

To get all students data but excluding the id .We use the command **db.students.find({},{-id:0});**

```
db> db.students.find({},{_id:0});
[
  {
    name: 'Student 948',
    age: 19,
    courses: "['English', 'Computer Science', 'Physics', 'Mathematics']",
    gpa: 3.44,
    home_city: 'City 2',
    blood_group: 'O+',
    is_hotel_resident: true
  },
  {
    name: 'Student 157',
    age: 20,
    courses: "['Physics', 'English']",
    gpa: 2.27,
    home_city: 'City 4',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    name: 'Student 316',
    age: 20,
    courses: "['Physics', 'Computer Science', 'Mathematics', 'History']",
    gpa: 2.32,
    blood_group: 'B+',
    is_hotel_resident: true
  },
  {
    name: 'Student 346',
    age: 25,
    courses: "['Mathematics', 'History', 'English']",
    gpa: 3.31,
    home_city: 'City 8',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    name: 'Student 930',
    age: 25,
    courses: "['English', 'Computer Science', 'Mathematics', 'History']",
    gpa: 3.63,
    home_city: 'City 3',
    blood_group: 'A-',
    is_hotel_resident: true
  },
  {
    name: 'Student 305',
    age: 24,
    courses: "['History', 'Physics', 'Computer Science', 'Mathematics']",
    gpa: 3.4,
    home_city: 'City 6',
    blood_group: 'O+',
    is_hotel_resident: true
  },
  {
    name: 'Student 268',
    age: 21,
    courses: "['Mathematics', 'History', 'Physics']",
    gpa: 3.98,
```

Here students data is shown without id.

Retrieving specific Fields from Nested Objects:

```
db> db.students.find({}, { name: 1, courses: { $slice: 2 } });
[
  {
    _id: ObjectId('66682c12c4b3310818df6409'),
    name: 'Student 948',
    courses: "['English', 'Computer Science', 'Physics', 'Mathematics']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df640a'),
    name: 'Student 157',
    courses: "['Physics', 'English']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df640b'),
    name: 'Student 316',
    courses: "['Physics', 'Computer Science', 'Mathematics', 'History']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df640c'),
    name: 'Student 346',
    courses: "['Mathematics', 'History', 'English']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df640d'),
    name: 'Student 930',
    courses: "['English', 'Computer Science', 'Mathematics', 'History']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df640e'),
    name: 'Student 305',
    courses: "['History', 'Physics', 'Computer Science', 'Mathematics']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df640f'),
    name: 'Student 268',
    courses: "['Mathematics', 'History', 'Physics']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df6410'),
    name: 'Student 563',
    courses: "['Mathematics', 'English']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df6411'),
    name: 'Student 440',
    courses: "['History', 'Physics', 'Computer Science']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df6412'),
    name: 'Student 536',
    courses: "['History', 'Physics', 'English', 'Mathematics']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df6413'),
    name: 'Student 256',
    courses: "['Computer Science', 'Mathematics', 'History', 'English']"
  },
  {
    _id: ObjectId('66682c12c4b3310818df6414'),
    name: 'Student 177',
```

Here only students name and the second course is obtained.

Limit

- The limit operator is used with the find method.
- It's chained after the filter criteria or any sorting operations.
- Syntax: db.collection.find({filter}, {projection}).limit(number)

To get first 5 document:

```
db> db.students.find({},{_id:0}).limit(5);
[
  {
    name: 'Student 948',
    age: 19,
    courses: "['English', 'Computer Science', 'Physics', 'Mathematics']",
    gpa: 3.44,
    home_city: 'City 2',
    blood_group: 'O+',
    is_hotel_resident: true
  },
  {
    name: 'Student 157',
    age: 20,
    courses: "['Physics', 'English']",
    gpa: 2.27,
    home_city: 'City 4',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    name: 'Student 316',
    age: 20,
    courses: "['Physics', 'Computer Science', 'Mathematics', 'History']",
    gpa: 2.32,
    blood_group: 'B+',
    is_hotel_resident: true
  },
  {
    name: 'Student 346',
    age: 25,
    courses: "['Mathematics', 'History', 'English']",
    gpa: 3.31,
    home_city: 'City 8',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    name: 'Student 930',
    age: 25,
    courses: "['English', 'Computer Science', 'Mathematics', 'History']",
    gpa: 3.63,
    home_city: 'City 3',
    blood_group: 'A-',
    is_hotel_resident: true
  }
]
db>
```

In this snapshot we got first five documents of the students data.

Selectors

Selectors in MongoDB are used to specify criteria for querying, updating, or deleting documents in a collection. They are part of MongoDB's powerful query language, enabling you to precisely target documents based on various conditions.

Comparsion gt It:

To find all the students with age greater than 10. We use the command **db.students.find({age:{$gt:10}});**

```
]
db> db.students.find({age:{$gt:10}});
[
  {
    _id: ObjectId('66682c12c4b3310818df6409'),
    name: 'Student 948',
    age: 19,
    courses: "['English', 'Computer Science', 'Physics', 'Mathematics']",
    gpa: 3.44,
    home_city: 'City 2',
    blood_group: 'O+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640a'),
    name: 'Student 157',
    age: 20,
    courses: "['Physics', 'English']",
    gpa: 2.27,
    home_city: 'City 4',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640b'),
    name: 'Student 316',
    age: 20,
    courses: "['Physics', 'Computer Science', 'Mathematics', 'History']",
    gpa: 2.32,
    blood_group: 'B+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640c'),
    name: 'Student 346',
    age: 25,
    courses: "['Mathematics', 'History', 'English']",
    gpa: 3.31,
    home_city: 'City 8',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640d'),
    name: 'Student 930',
    age: 25,
    courses: "['English', 'Computer Science', 'Mathematics', 'History']",
    gpa: 3.63,
    home_city: 'City 3',
    blood_group: 'A-',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640e'),
    name: 'Student 305',
    age: 24,
    courses: "['History', 'Physics', 'Computer Science', 'Mathematics']",
    gpa: 3.4,
    home_city: 'City 6',
    blood_group: 'O+',
    is_hotel_resident: true
  },
```

Operators

Logical operators:

$and operator:

Each condition is defined that must meet for the document ti be selected.

$or:

Each condition is defined atleast one of its must be matched to return a document.

AND Operator:

To find the students from city 3 with blood group “A+”

```
db> db.students.find({
... $and:[
... {home_city:"City 3"},
... {blood_group:"A+"}
... ]
... });
[
  {
    _id: ObjectId('66682c12c4b3310818df641a'),
    name: 'Student 172',
    age: 25,
    courses: "['English', 'History', 'Physics', 'Mathematics']",
    gpa: 2.46,
    home_city: 'City 3',
    blood_group: 'A+',
    is_hotel_resident: false
  },
  {
    _id: ObjectId('66682c12c4b3310818df643e'),
    name: 'Student 959',
    age: 24,
    courses: "['History', 'Computer Science']",
    gpa: 3.43,
    home_city: 'City 3',
    blood_group: 'A+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df64ce'),
    name: 'Student 918',
    age: 19,
    courses: "['Physics', 'Computer Science']",
    gpa: 3.92,
    home_city: 'City 3',
    blood_group: 'A+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df6515'),
    name: 'Student 728',
    age: 24,
    courses: "['Mathematics', 'Physics', 'English']",
    gpa: 3.95,
    home_city: 'City 3',
    blood_group: 'A+',
    is_hotel_resident: true
  }
]
```

Here we got the details of the students from city 3 with blood group A+.

OR Operator:

To find the students who are hotel residents OR have a gpa less than 2.

```
]
db> db.students.find({
... $or:[
... {is_hotel_resident:true},
... {gpa:{$lt:2.0}}
... ]
... });
[
  {
    _id: ObjectId('66682c12c4b3310818df6409'),
    name: 'Student 948',
    age: 19,
    courses: "['English', 'Computer Science', 'Physics', 'Mathematics']",
    gpa: 3.44,
    home_city: 'City 2',
    blood_group: 'O+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640a'),
    name: 'Student 157',
    age: 20,
    courses: "['Physics', 'English']",
    gpa: 2.27,
    home_city: 'City 4',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640b'),
    name: 'Student 316',
    age: 20,
    courses: "['Physics', 'Computer Science', 'Mathematics', 'History']",
    gpa: 2.32,
    blood_group: 'B+',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640c'),
    name: 'Student 346',
    age: 25,
    courses: "['Mathematics', 'History', 'English']",
    gpa: 3.31,
    home_city: 'City 8',
    blood_group: 'O-',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640d'),
    name: 'Student 930',
    age: 25,
    courses: "['English', 'Computer Science', 'Mathematics', 'History']",
    gpa: 3.63,
    home_city: 'City 3',
    blood_group: 'A-',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('66682c12c4b3310818df640e'),
    name: 'Student 305',
    age: 24,
    courses: "['History', 'Physics', 'Computer Science', 'Mathematics']",
    gpa: 3.4,
```

Here students with gpa less than 2.0 is obtained.

Bitwise types

Bitwise

Name	Description
`$bitsAllClear`	Matches numeric or binary values in which a set of bit positions *all* have a value of `0`.
`$bitsAllSet`	Matches numeric or binary values in which a set of bit positions *all* have a value of `1`.
`$bitsAnyClear`	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of `0`.
`$bitsAnySet`	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of `1`.

Query

Querying in MongoDB involves using selectors to filter documents in a collection.

To find students with both lobby and campus permission

db.students_permission.find({permissions:{$bitsAllSet:[LOBBY_ PER MISSION, CAMPUS-PERMISSION]}});

First we need to define bit positions for permisssions (collection)so we defines

const LOBBY_PERMISSION=1; const CAMPUS-PERMISSION=2

```
b> db.students_permission.find({permissions:{$bitsAllSet:[LOBBY_PERMISS
ON,CAMPUS_PERMISSION]}});
[
  {
    _id: ObjectId('6663ff4286ef416122dcfcd5'),
    name: 'George',
    age: 21,
    permissions: 6
  },
  {
    _id: ObjectId('6663ff4286ef416122dcfcd6'),
    name: 'Henry',
    age: 27,
    permissions: 7
  },
  {
    _id: ObjectId('6663ff4286ef416122dcfcd7'),
    name: 'Isla',
    age: 18,
    permissions: 6
  }
]
b>
```

Geospatial Query

Geospatial queries in MongoDB allow you to work with locationbased data, such as finding documents within a certain distance from a point, within a polygon, or near other shapes. MongoDB supports various geospatial operations and indexing types to efficiently handle these queries.

Here to find a location:

```
db> db.locations.find({
... location:{
... $geoWithin:{
... $centerSphere:[[-74.005,40.712],0.006213776]
... }
... }
... });
[
  {
    _id: 1,
    name: 'Coffee Shop A',
    location: { type: 'Point', coordinates: [ -73.985, 40.748 ] }
  },
  {
    _id: 2,
    name: 'Restaurant B',
    location: { type: 'Point', coordinates: [ -74.009, 40.712 ] }
  },
  {
    _id: 5,
    name: 'Park E',
    location: { type: 'Point', coordinates: [ -74.006, 40.705 ] }
  }
]
```

CLASS 5 : Projection Operators

Projection operators specify the fields returned by an operation.

Projection operators:

- $
- $elemMatch
- $slice

$

The positional **$** operator limits the contents of an to return the first <array> element that matches the query condition on the array.

$elemMatch

The **$elemMatch** projection operator takes an explicit condition argument. This allows you to project based on a condition not in the query, or if you need to project based on multiple fields in the array's embedded documents

$slice

The **$slice** projection operator specifies the number of elements in an array to return in the query result.

Retrieve Name, Age, and GPA

```
db> db.candidates.find({},{name:1,age:1,gpa:1});
[
  {
    _id: ObjectId('669923074a062f6e859a18cf'),
    name: 'Alice Smith',
    age: 20,
    gpa: 3.4
  },
  {
    _id: ObjectId('669923074a062f6e859a18d0'),
    name: 'Bob Johnson',
    age: 22,
    gpa: 3.8
  },
  {
    _id: ObjectId('669923074a062f6e859a18d1'),
    name: 'Charlie Lee',
    age: 19,
    gpa: 3.2
  },
  {
    _id: ObjectId('669923074a062f6e859a18d2'),
    name: 'Emily Jones',
    age: 21,
    gpa: 3.6
  },
  {
    _id: ObjectId('669923074a062f6e859a18d3'),
    name: 'David Williams',
    age: 23,
    gpa: 3
  },
  {
    _id: ObjectId('669923074a062f6e859a18d4'),
    name: 'Fatima Brown',
    age: 18,
    gpa: 3.5
  },
  {
    _id: ObjectId('669923074a062f6e859a18d5'),
    name: 'Gabriel Miller',
    age: 24,
    gpa: 3.9
  },
  {
    _id: ObjectId('669923074a062f6e859a18d6'),
    name: 'Hannah Garcia',
    age: 20,
    gpa: 3.3
  },
  {
    _id: ObjectId('669923074a062f6e859a18d7'),
    name: 'Isaac Clark',
    age: 22,
    gpa: 3.7
  },
  {
    _id: ObjectId('669923074a062f6e859a18d8'),
    name: 'Jessica Moore',
    age: 19,
    gpa: 3.1
  },
  {
    _id: ObjectId('669923074a062f6e859a18d9'),
    name: 'Kevin Lewis',
    age: 21,
    gpa: 4
  },
```

Here only name, age and gpa are displayed.

Variation: Exclude fields

```
]
db> db.candidates.find({},{_id:0,course:0});
[
  {
    name: 'Alice Smith',
    age: 20,
    courses: [ 'English', 'Biology', 'Chemistry' ],
    gpa: 3.4,
    home_city: 'New York City',
    blood_group: 'A+',
    is_hotel_resident: true
  },
  {
    name: 'Bob Johnson',
    age: 22,
    courses: [ 'Computer Science', 'Mathematics', 'Physics' ],
    gpa: 3.8,
    home_city: 'Los Angeles',
    blood_group: 'O-',
    is_hotel_resident: false
  },
  {
    name: 'Charlie Lee',
    age: 19,
    courses: [ 'History', 'English', 'Psychology' ],
    gpa: 3.2,
    home_city: 'Chicago',
    blood_group: 'B+',
    is_hotel_resident: true
  },
  {
    name: 'Emily Jones',
    age: 21,
    courses: [ 'Mathematics', 'Physics', 'Statistics' ],
    gpa: 3.6,
    home_city: 'Houston',
    blood_group: 'AB-',
    is_hotel_resident: false
  },
  {
    name: 'David Williams',
    age: 23,
    courses: [ 'English', 'Literature', 'Philosophy' ],
    gpa: 3,
    home_city: 'Phoenix',
    blood_group: 'A-',
    is_hotel_resident: true
  },
  {
    name: 'Fatima Brown',
    age: 18,
    courses: [ 'Biology', 'Chemistry', 'Environmental Science' ],
    gpa: 3.5,
    home_city: 'San Antonio',
    blood_group: 'B+',
    is_hotel_resident: false
  },
  {
    name: 'Gabriel Miller',
    age: 24,
    courses: [ 'Computer Science', 'Engineering', 'Robotics' ],
    gpa: 3.9,
    home_city: 'San Diego',
    blood_group: 'O+',
    is_hotel_resident: true
  },
  {
```

Here _id, courses are exluded .

$slice

2. Retrieve all candidates with first two courses

```
db> db.candidates.find({},{name:1,courses:{$slice:2}});
[
  {
    _id: ObjectId('669923074a062f6e859a18cf'),
    name: 'Alice Smith',
    courses: [ 'English', 'Biology' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d0'),
    name: 'Bob Johnson',
    courses: [ 'Computer Science', 'Mathematics' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d1'),
    name: 'Charlie Lee',
    courses: [ 'History', 'English' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d2'),
    name: 'Emily Jones',
    courses: [ 'Mathematics', 'Physics' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d3'),
    name: 'David Williams',
    courses: [ 'English', 'Literature' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d4'),
    name: 'Fatima Brown',
    courses: [ 'Biology', 'Chemistry' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d5'),
    name: 'Gabriel Miller',
    courses: [ 'Computer Science', 'Engineering' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d6'),
    name: 'Hannah Garcia',
    courses: [ 'History', 'Political Science' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d7'),
    name: 'Isaac Clark',
    courses: [ 'English', 'Creative Writing' ]
  },
  {
    _id: ObjectId('669923074a062f6e859a18d8'),
    name: 'Jessica Moore',
    courses: [ 'Biology', 'Ecology' ]
  },
  {
```

Return only the name field and the first 2 elements of the courses array for each document.

CLASS 6 : Aggregation Operators

- Aggregation means grouping together
- For ex : sum, avg, min, max

Syntax :

db.collection.aggregate(<AGGREGATE OPERATION>

Types

Expression Type	**Description**	**Syntax**
Accumulators	Perform calculations on entire groups of documents	
* $sum	Calculates the sum of all values in a numeric field within a group.	"$fieldName": { $sum: "$fieldName" }
* $avg	Calculates the average of all values in a numeric field within a group.	"$fieldName": { $avg: "$fieldName" }
* $min	Finds the minimum value in a field within a group.	"$fieldName": { $min:

		"$fieldName" }
* $max	Finds the maximum value in a field within a group.	"$fieldName": { $max: "$fieldName" }
* $push	Creates an array containing all unique or duplicate values from a field	"$arrayName": { $push: "$fieldName" }
* $addToSet	Creates an array containing only unique values from a field within a group.	"$arrayName": { $addToSet: "$fieldName" }
* $first	Returns the first value in a field within a group (or entire collection).	"$fieldName": { $first: "$fieldName" }
* $last	Returns the last value in a field within a group (or entire collection).	"$fieldName": { $last: "$fieldName" }

1. Average GPA of all students

```
db> db.students.aggregate([
... {$group: {_id:null,averageGPA:{$avg:"$gpa"}}}
... ]);
[ { _id: null, averageGPA: 2.98556 } ]
db>
```

_id: null:

Sets the group identifier to null (optional, as there's only one group in this case)

averageGPA: Calculates the average value of the "gpa" field using the $avg operator.

2.Minimum and Maximum Age:

```
db> db.students.aggregate([
... {$group:{_id:null,minAge:{$min:"$age"},maxAge:{$max:"$age"}}}
... ]);
[ { _id: null, minAge: 18, maxAge: 25 } ]
db>
```

minAge:

Uses the $min operator to find the minimum value in the "age" field. maxAge:

Uses the $max operator to find the maximum value in the "age" field.

This shows only min age and max age.

3. Average GPA of all home cities:

```
db> db.students.aggregate([
... {$group:{_id:"$home_city",averageGPA:{$avg:"$gpa"}}}
... ]);
[
  { _id: 'City 7', averageGPA: 2.8479310344827 59 },
  { _id: 'City 3', averageGPA: 3.0100000000000002 },
  { _id: 'City 8', averageGPA: 3.117419354838 71 },
  { _id: 'City 6', averageGPA: 2.8969444444444448 },
  { _id: 'City 4', averageGPA: 2.8251851851851852 },
  { _id: 'City 1', averageGPA: 3.0038235294117 65 },
  { _id: 'City 10', averageGPA: 2.9352272727272 73 },
  { _id: 'City 5', averageGPA: 3.0607499999999996 },
  { _id: 'City 9', averageGPA: 3.1174358974358976 },
  { _id: null, averageGPA: 2.9784313725490197 },
  { _id: 'City 2', averageGPA: 3.019696969696 97 }
]
```

Group the documents in the students collection by the home_city field.

For each group, calculate the average GPA (gpa) and store it in a field called averageGPA.

The result will be a set of documents where each document represents a unique home_city and the corresponding average GPA of students from that city.

4. Pushing all courses into a single array:

```
db> db.students.aggregate([
... {$unwind:"$courses"},
... {$group:{_id:null,allCourses:{$push:"$courses"}}},
... {$project:{_id:0,allCourses:1}}
... ]);
[
  {
    allCourses: [
      "['English', 'Computer Science', 'Physics', 'Mathematics']",
      "['Physics', 'English']",
      "['Physics', 'Computer Science', 'Mathematics', 'History']",
      "['Mathematics', 'History', 'English']",
      "['English', 'Computer Science', 'Mathematics', 'History']",
      "['History', 'Physics', 'Computer Science', 'Mathematics']",
      "['Mathematics', 'History', 'Physics']",
      "['Mathematics', 'English']",
      "['History', 'Physics', 'Computer Science']",
      "['History', 'Physics', 'English', 'Mathematics']",
      "['Computer Science', 'Mathematics', 'History', 'English']",
      "['Mathematics', 'Computer Science', 'Physics']",
      "['Mathematics', 'Computer Science']",
      "['History', 'Physics', 'Computer Science']",
      "['English', 'History']",
      "['History', 'Physics', 'Mathematics']",
      "['English', 'History', 'Physics', 'Computer Science']",
      "['English', 'History', 'Physics', 'Mathematics']",
      "['English', 'Physics']",
      "['Computer Science', 'Physics', 'History', 'Mathematics']",
      "['Physics', 'Computer Science', 'English']",
      "['Computer Science', 'English', 'History']",
      "['Mathematics', 'English', 'Computer Science', 'Physics']",
      "['Computer Science', 'Physics', 'Mathematics', 'History']",
      "['Physics', 'Computer Science', 'English', 'Mathematics']",
      "['History', 'Computer Science']",
      "['Computer Science', 'History', 'Physics', 'English']",
      "['History', 'English', 'Computer Science', 'Mathematics']",
      "['English', 'Mathematics', 'Computer Science', 'Physics']",
      "['History', 'Computer Science', 'Mathematics', 'English']",
      "['History', 'Physics', 'Computer Science']",
      "['Mathematics', 'Computer Science']",
      "['Mathematics', 'Computer Science']",
      "['History', 'Computer Science', 'Physics', 'Mathematics']",
      "['Computer Science', 'English', 'Physics', 'History']",
      "['History', 'Mathematics', 'Physics', 'English']",
      "['Computer Science', 'Mathematics', 'English', 'Physics']",
      "['English', 'Mathematics', 'Computer Science']",
      "['English', 'History']",
      "['Mathematics', 'History', 'Physics']",
      "['English', 'Physics', 'Computer Science', 'History']",
      "['History', 'Computer Science']",
      "['English', 'Physics', 'Mathematics', 'Computer Science']",
      "['English', 'Physics']",
      "['English', 'Computer Science', 'Physics']",
      "['Mathematics', 'Physics', 'Computer Science', 'English']",
      "['Physics', 'Computer Science']",
      "['History', 'Mathematics', 'Physics']",
      "['Computer Science', 'English', 'History']",
      "['English', 'History', 'Computer Science']",
      "['Physics', 'History', 'Mathematics']",
      "['Computer Science', 'History']",
      "['English', 'History', 'Mathematics', 'Computer Science']",
      "['History', 'Computer Science']",
      "['Physics', 'English', 'Mathematics', 'History']",
      "['History', 'English', 'Physics', 'Computer Science']",
```

The final output is a single document with an allCourses array containing all the courses from all students.

CLASS 7 : Aggregation pipleline

Allow you to transform and analyze your data in a pipeline format. An aggregation pipeline consists of multiple stages that process documents, each stage performing an operation on the documents and passing the results to the next stage.

- `$match` : Filters documents based on a condition.
- `$group` : Groups documents by a field and performs aggregations like `$avg` (average) and `$sum` (sum).
- `$sort` : Sorts documents in a specified order (ascending or descending).
- `$project` : Selects specific fields to include or exclude in the output documents.
- `$skip` : Skips a certain number of documents from the beginning of the results.
- `$limit` : Limits the number of documents returned.
- `$unwind` : Deconstructs an array into separate documents for each element.

Benefits of Aggregation Pipeline:

- **Flexible and powerful**:

  Allows complex transformations and analyses.

- **Scalable**:

  Can handle large datasets efficiently.

- **Modular**:

  Each stage in the pipeline can be developed and tested independently.

1.To find students with age greater than 23, sorted by age in descending order and only return name and age.

```
db> db.students6.aggregate([
... {$match:{age:{$gt:23}}},
... {$sort:{age:-1}},
... {$project:{_id:0,name:1,age:1}}
... ]);
[ { name: 'Charlie', age: 28 }, { name: 'Alice', age: 25 } ]
db>
```

Here students older than 23 sorted by age in descending order and displays only name and age.

2.To find students with age greater than 23, sorted by age in ascending order and only return name and age.

```
db> db.students6.aggregate([ { $match: { age: { $lt: 23 } } }, { $sort: { age: -1 }
 }, { $project: { _id: 0, name: 1, age: 1 } }] );
[ { name: 'Bob', age: 22 }, { name: 'David', age: 20 } ]
db>
```

Here students younger than 23 sorted by age in ascending order and displays only name and age.

3.Group students by major, calculate average age and total number of students in each major.

```
db> db.students6.aggregate([
...   { $group: { _id: "$major", averageAge: { $avg: "$age" }, totalStudents: { $sum: 1 } } }
... ])
[
  { _id: 'Mathematics', averageAge: 22, totalStudents: 1 },
  { _id: 'English', averageAge: 28, totalStudents: 1 },
  { _id: 'Computer Science', averageAge: 22.5, totalStudents: 2 },
  { _id: 'Biology', averageAge: 23, totalStudents: 1 }
]
```

Here average age and total number of students in each major is calculated.

4.Find students with an average score (from scores array) above 85 and skip the first document.

```
db> db.students6.aggregate([ { $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } } }, { $match: { averageScore: { $gt: 85 } } }, { $skip: 1 }] );
[ { name: 'David', averageScore: 93.33333333333333 } ]
db>
```

The $project stage calculates the average score for each student and includes only the name and averageScore fields.

The $match stage filters out students whose averageScore is not greater than 85.

The $skip stage skips the first document in the filtered result set, so only the subsequent documents are included in the final output.

5.Find students with an average score (from scores array) below 86 and skip the first 2 documents.

The $project stage calculates the average score for each student and includes only the name and averageScore fields.

The $match stage filters out students whose averageScore is not less than 86.

The $skip stage skips the first two documents in the filtered result set, resulting in no documents in the final output.

CLASS 8 : ACID & Indexes

ACID

- Atomicity
- Consistency
- Replication
- Sharding

Atomicity

Atomicity in MongoDB ensures that operations are executed as a complete unit, preventing partial updates and maintaining data consistency. Single document operations are atomic by default, while multi-document transactions provide atomicity across multiple documents and collections.

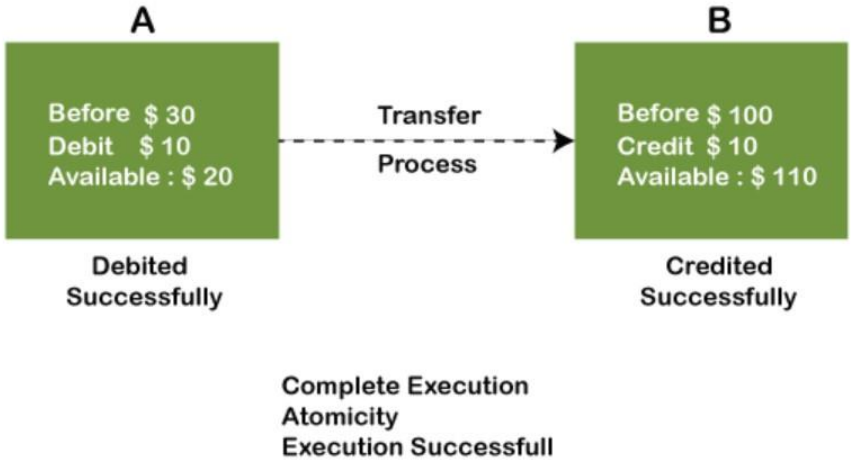
1. Data Consistency:

- **Single Document Updates**: Ensures that updates to a single document are completed fully or not at all, preventing partial updates that can lead to inconsistent states.
- **Multi-Document Transactions**: Starting with MongoDB 4.0, multi-document transactions ensure that a series of operations across multiple documents and

collections are either fully completed or fully rolled back, maintaining data consistency.

2. Complex Operations:

- **Banking Transactions**: In scenarios like transferring money between accounts, atomicity ensures that the debit from one account and the credit to another happen together. If either operation fails, neither account is updated.
- **Inventory Management**: Updating inventory levels and recording the transaction simultaneously ensures that the inventory data remains accurate and consistent.



3. Error Handling:

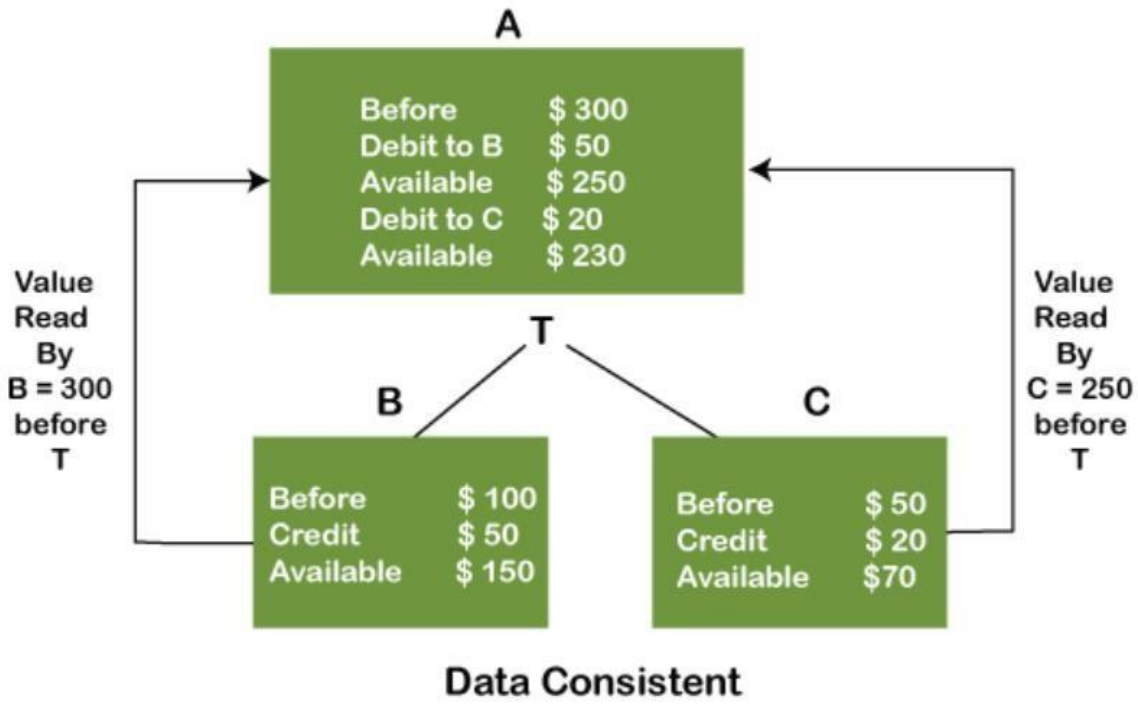
- Atomicity simplifies error handling in applications. Developers can handle errors by aborting transactions, knowing that no partial updates will remain in the database.

4. Data Integrity:

- Ensures that the database remains in a valid state, even in the event of system crashes or power failures. Unfinished transactions are not applied, preventing corrupt or incomplete data from being saved.

Consistency

Consistency ensures that the database remains in a valid and predictable state after any operation. This is achieved through schema validation, atomic operations, replica sets, indexes, and multidocument transactions. By maintaining consistency, MongoDB helps developers build reliable applications with accurate and trustworthy
data.



1. **Schema Validation**:

- MongoDB allows for schema validation rules that enforce specific constraints on the structure and content of documents within a collection. This ensures that any document inserted or updated adheres to the defined schema, maintaining data integrity.

2. **Atomic Operations**:

- As mentioned earlier, single-document operations in MongoDB are atomic, which helps maintain consistency. Multi-document transactions also provide atomicity, ensuring that a series of operations are either fully completed or not applied at all.

3. **Replica Sets:**

- MongoDB uses replica sets to ensure data redundancy and availability. Consistency is maintained through replication, where secondary members replicate data from the primary member. MongoDB offers various read preferences that can affect consistency, such as reading from the primary for strong consistency or from secondaries for eventual consistency.

4. **Indexes:**

- Indexes in MongoDB ensure that data remains consistent and queries return accurate results efficiently. Unique indexes, for instance, enforce that no two documents have the same value for a specific field, maintaining data consistency.

5. **Transactions:**

- Starting with MongoDB 4.0, support for multi-document transactions ensures that operations across multiple documents and collections are consistent. If any part of a transaction fails, the entire transaction is rolled back, preserving the consistent state of the database.

Eventual Consistency

Eventual consistency in MongoDB refers to a consistency model where updates to a distributed database are propagated to all nodes asynchronously, ensuring that all nodes will eventually become consistent, given enough time. This model is particularly relevant in distributed systems and can be seen in MongoDB replica sets when read preferences are set to read from secondary members.

Key Concepts:

1. **Primary and Secondary Nodes:**
    - In a MongoDB replica set, one node is the primary (which receives all write operations), and the others are secondary nodes (which replicate the data from the primary).
2. **Replication Lag:**
    - There can be a delay (replication lag) between when the data is written to the primary and when it is replicated to the secondaries. During this time, the secondary nodes may not have the most recent data.

3. **Read Preferences:**

   - MongoDB allows applications to specify read preferences. If the read preference is set to secondary, secondaryPreferred, or nearest, clients may read stale data because secondaries might not have the latest updates.

Example:

User A posts a photo:

- The photo is written to the primary node.
- The write operation will eventually be replicated to the secondary nodes.

User B views User A's profile:

- User B reads from one of the secondary nodes.
- Due to replication lag, the new post might not be immediately visible to User B.

In an Instagram-like application, eventual consistency allows the system to be highly available and responsive. Users can continue reading from secondary nodes even if they might temporarily see slightly outdated information. This is acceptable in scenarios where seeing the most up-to-date information is not critical for every user interaction.

Isolation

Isolation refers to how the database handles concurrent operations, ensuring data consistency and integrity. MongoDB uses a form of isolation called multi-granularity locking (MGL) that allows operations to lock at different levels of granularity, such as global, database, collection, or document levels. This allows MongoDB to provide better performance and concurrency compared to databases that only support row-level locking or table-level locking.

1.Document-level Locking: MongoDB uses document-level locking, meaning each document in a collection can be locked independently. This improves concurrency since operations on different documents can proceed simultaneously without waiting for locks to be released on other documents.

2.Multi-granularity Locking (MGL): MongoDB employs MGL, which allows it to lock at various levels:

Global: Affects the entire MongoDB instance.

Database: Affects a specific database.

Collection: Affects a specific collection.

Document: Affects a specific document.

3.Write Concern and Read Concern:

Write Concern: Controls the acknowledgment of write operations. It specifies the level of acknowledgment requested from MongoDB for write operations. Higher write concern levels ensure greater durability but might impact performance.

Read Concern: Controls the consistency and isolation properties of the data read from a replica set. It allows you to specify the level of isolation for read operations, ensuring that read operations return data that meets the specified consistency level.

4.Transactional Isolation:

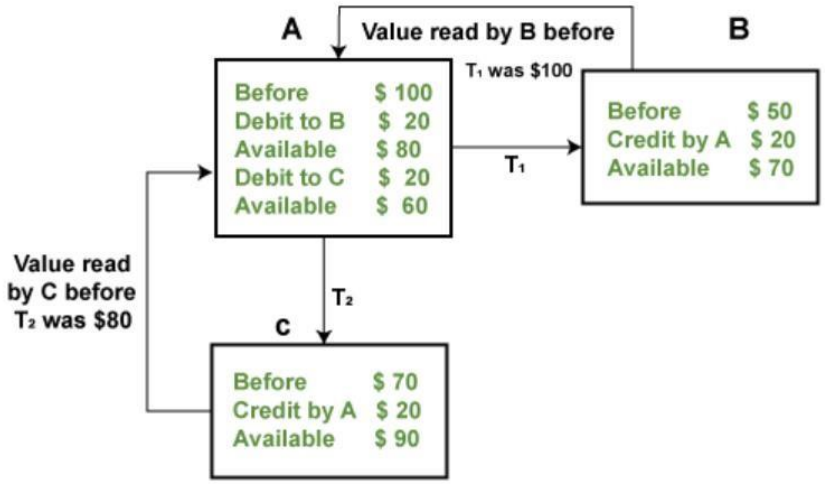
- MongoDB supports multi-document ACID transactions starting from version 4.0. Transactions provide snapshot isolation, ensuring that all reads within a transaction see a consistent snapshot of the data as it was at the start of the transaction.
- Transactions can span multiple documents, collections, databases, and even shards in a sharded cluster.

5.Isolation Levels:

- MongoDB provides snapshot isolation within transactions. This ensures that the data read during a transaction will not change until the transaction completes.
- Outside of transactions, MongoDB operations provide isolation at the document level, ensuring that operations on a single document are atomic and isolated from operations on other documents.

Example:

Suppose you have a banking application where you need to transfer money between two accounts. Using MongoDB transactions, you can ensure that the debit from one account and the credit to another account are performed atomically, preventing any inconsistencies that could arise from concurrent operations.



Benefits

1.Data Consistency: Ensures that operations on the database do not leave it in an inconsistent state.

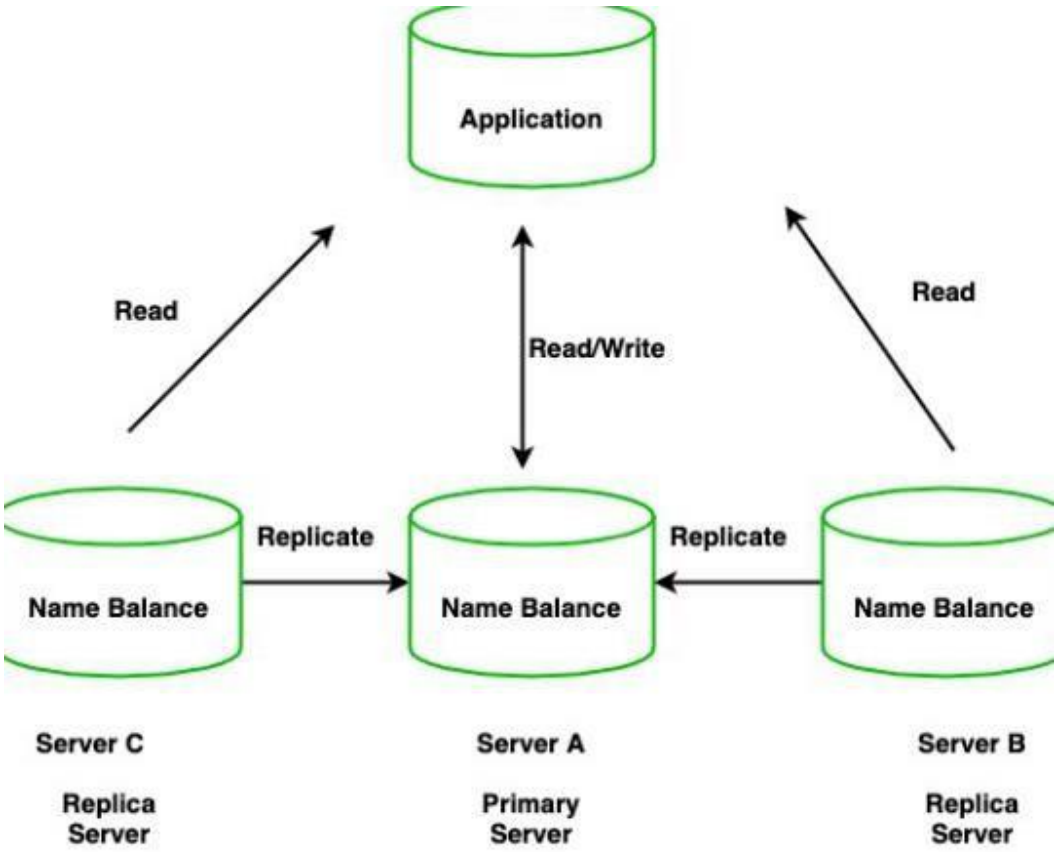
2.Atomic Operations: Guarantees that multiple operations either all succeed or all fail, maintaining atomicity.

3.Concurrency Control: Allows multiple operations to be performed concurrently without interference, improving performance.

4.Error Handling: Simplifies error handling by allowing transactions to be rolled back in case of failures.

Replication (Master-Salve)

Replication in MongoDB is a process that allows data to be copied and distributed across multiple servers, ensuring high availability, redundancy, and disaster recovery. This mechanism is crucial for maintaining data reliability and ensuring that the system remains operational even in the event of hardware failures or other issues.



Benefits of Replication:

1. **High Availability**:

   - Replication ensures that data remains available even if one or more nodes fail. o Automatic failover allows secondary nodes to take over as the primary node without manual intervention.

2. **Data Redundancy**:

   - Multiple copies of data are maintained across different nodes. o This redundancy protects against data loss due to hardware failures.

3. **Load Balancing**:

   - Read operations can be distributed across secondary nodes, reducing the load on the primary node. o This improves performance and responsiveness for readheavy applications.

4. **Disaster Recovery**:

   - Replica sets can span multiple data centers or geographic regions. o This geographic distribution ensures that data is protected against regional failures or disasters.

Sharding

Sharding is a method for distributing data across multiple servers, or shards, to ensure scalability and manage large data sets. It helps to distribute the load and handle large volumes of data by dividing it into smaller, more manageable pieces. Each shard holds a subset of the sharded data, and collectively the shards form a single logical database.

1.Shard:

- A shard is a single MongoDB instance or replica set that holds a portion of the sharded data.

2.**Shard Key**:

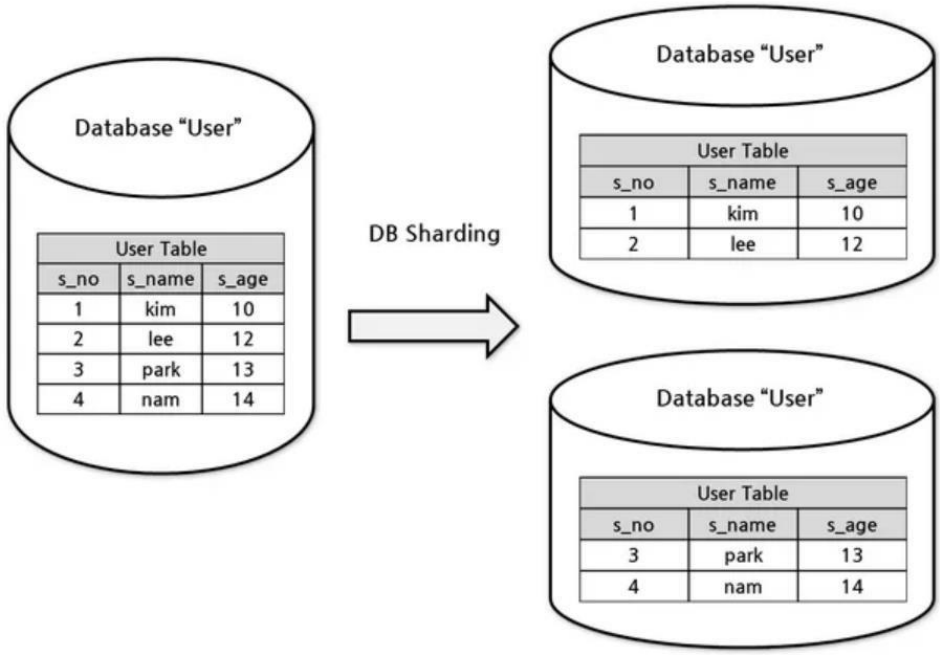
- The shard key is a field or set of fields that determines how data is distributed across the shards.
- It is crucial to choose an appropriate shard key to ensure even data distribution and avoid bottlenecks.

3.**Config Server**:

- Config servers store metadata and configuration settings for the sharded cluster.
- They keep track of the distribution of data and the state of the cluster.

4.**Query Router (mongos)**:

- The query router, or mongos, directs queries from client applications to the appropriate shards.
- It uses the metadata stored in the config servers to determine which shards to query.



Benefits of Sharding:

1. **Horizontal Scalability**:
    - Sharding allows a database to scale out horizontally by adding more servers to handle increased load and data volume.
    - This helps in managing very large data sets efficiently.

2. **Improved Performance**:

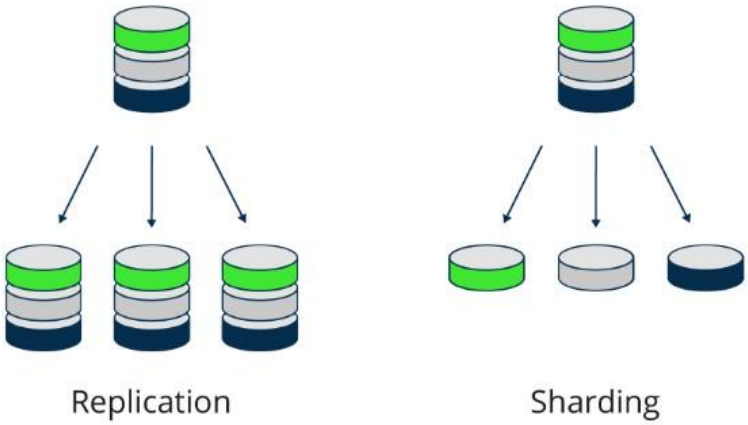
   - By distributing data and queries across multiple shards, sharding can significantly improve read and write performance. o It helps to avoid performance bottlenecks that occur when a single server handles all operations.

3. **High Availability**:

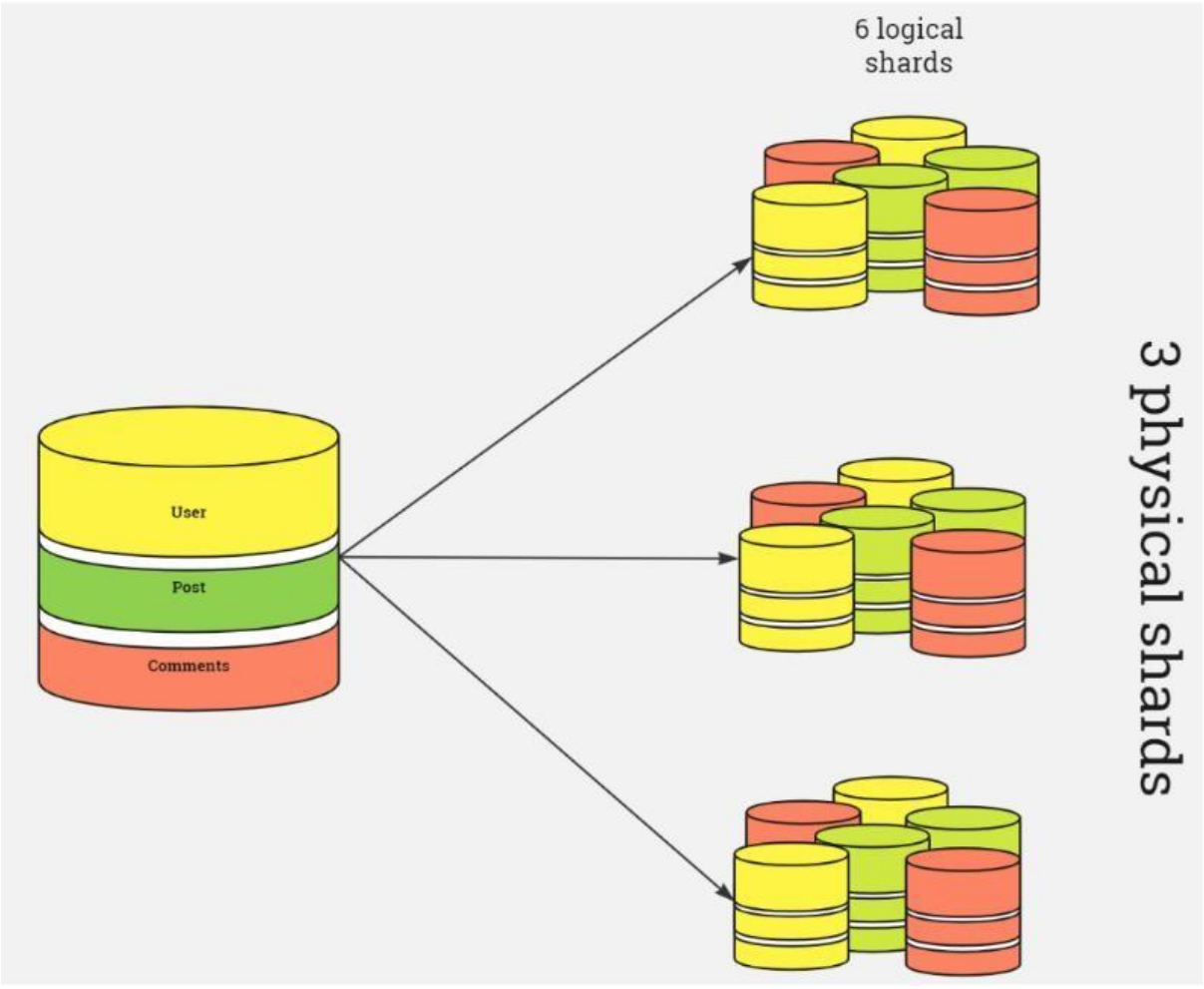
   - Sharded clusters often use replica sets for each shard, providing high availability and redundancy. o This ensures that data remains available even if some nodes fail.

Replication v/s Sharding

Feature	Replication	Sharding
Purpose	High availability and redundancy	Scalability and distribution of data
Data Distribution	Copies of the same data across multiple nodes	Different subsets of data across multiple nodes
Primary Component	Replica Set	Sharded Cluster
Nodes	Primary and secondary nodes	Shards, Config Servers, and Query Routers (mongos)
Write Operations	Only primary node accepts writes	Writes distributed based on shard key
Read Operations	Can be served by any node (primary or secondary)	Directed by mongos to appropriate shards
Fault Tolerance	High, due to data redundancy	Moderate, relies on replication within shards
Automatic Failover	Yes, automatic election of new primary	Depends on replica sets within shards
Data Consistency	Strong consistency by default, tunable with read and write concerns	Ensured within shards, requires additional configuration across shards
Load Balancing	Not inherently load balanced, can read from secondaries to distribute read load	Yes, data is distributed across shards
Use Case	High availability and disaster recovery	Large datasets and high throughput operations
Setup Complexity	Moderate	High, requires careful planning and configuration
Maintenance	Easier, mostly involves monitoring replication status	More complex, involves monitoring data distribution and balancing
Query Complexity	Simple, queries directed to primary or secondaries	More complex, queries routed to appropriate shards by mongos
Scalability	Limited to vertical scaling (more powerful servers)	Horizontal scaling (adding more shards)



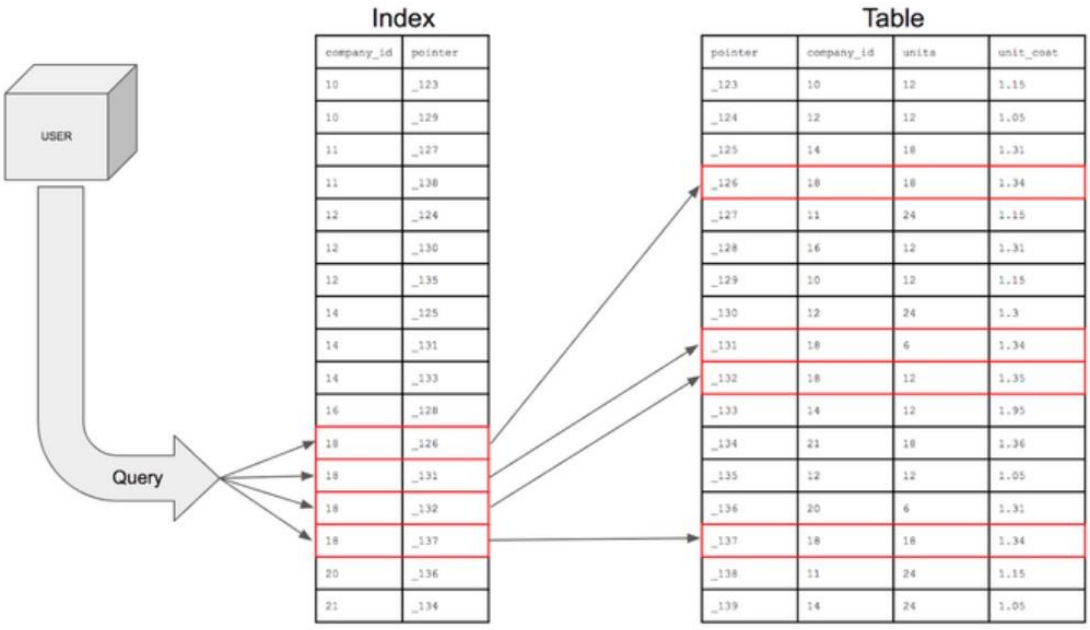
Replication + Sharding



Aspect	Replication + Sharding
Purpose	Combines data distribution with fault tolerance.
Components	Sharded Cluster: Multiple shards, each being a replica set.
Data Distribution	Data is distributed across shards, and each shard is replicated.
Failover	Failover occurs both at the shard level and within individual shards.
Query Routing	Queries are routed by the mongos instances, which direct them to the appropriate shard.
Configuration	Requires configuration of both replication (replica sets) and sharding.

Indexes

Indexes in MongoDB are crucial for optimizing query performance and improving data retrieval speed. They help MongoDB efficiently find and sort data, which is especially important for large datasets and complex queries.



An index is a data structure that improves the speed of data retrieval operations on a collection.

MongoDB indexes work similarly to indexes in traditional databases by allowing the database engine to quickly locate and access the desired data.

Types

1.Single Field Index: Indexes a single field of a document. It is the simplest form of index.

2.Compound Index: Indexes multiple fields. Useful for queries that filter or sort on multiple fields.

3.Multikey Index: Indexes fields that contain arrays. This allows MongoDB to index array elements.

4.Text Index: Indexes text content for text search queries. Supports searching for words or phrases.

5.Geospatial Index: Indexes geographic location data to support geospatial queries.

6.Hashed Index: Indexes a hashed value of the field. Useful for sharding.

Index Options:

- **Unique**: Ensures that all values for a given field or set of fields are unique across documents.
- **Sparse**: Indexes only documents that contain the indexed field, ignoring documents that do not contain it.
- **TTL (Time-To-Live)**: Automatically removes documents from the collection after a certain period.

Uses

1. Speed Up Queries
2. Improve Performance of Aggregation Pipelines
3. Enable Efficient Full-Text Search
4. Support Geospatial Queries
5. Optimize Uniqueness Constraints