

BITWISE VALUE:

- In our example it's a 32 bit each bit representing different things
- Bitwise value 7 means all access 7->111

Bit 3	Bit 4	Bit 5
cafe	campus	lobby

BITWISE TYPES:

Bitwise

Name	Description
<code>\$bitsAllClear</code>	Matches numeric or binary values in which a set of bit positions <i>all</i> have a value of 0.
<code>\$bitsAllSet</code>	Matches numeric or binary values in which a set of bit positions <i>all</i> have a value of 1.
<code>\$bitsAnyClear</code>	Matches numeric or binary values in which <i>any</i> bit from a set of bit positions has a value of 0.
<code>\$bitsAnySet</code>	Matches numeric or binary values in which <i>any</i> bit from a set of bit positions has a value of 1.


QUERY:

In MongoDB, a query is a command used to retrieve documents from a collection that match certain criteria. Queries are written in MongoDB's query language and can include conditions, projections, and other modifiers to specify exactly which documents to retrieve. MongoDB queries can be complex, allowing for precise filtering, sorting, and aggregation of data.

```
db> const LOBBY_PERMISSION=1;
db> const CAMPUS_PERMISSION=2;
db> db.students_permission.find({
... permissions:{$bitsAllSet:[LOBBY_PERMISSION,CAMPUS_PERMISSION]}
... });
[
  {
    _id: ObjectId('66635182d29d811170a4e560'),
    name: 'George',
    age: 21,
    permissions: 6
  },
  {
    _id: ObjectId('66635182d29d811170a4e561'),
    name: 'Henry',
    age: 27,
    permissions: 7
  },
  {
    _id: ObjectId('66635182d29d811170a4e562'),
    name: 'Isla',
    age: 18,
    permissions: 6
  }
]
db>
```

1.Constant Defination:

```
const LOBBY_PERMISSION = 1;
const CAMPUS_PERMISSION = 2;
```



Two constants are defined: `LOBBY_PERMISSION` with a value of 1 and `CAMPUS_PERMISSION` with a value of 2. These constants are used to represent specific permissions.

2.Query Execution:

```
db.students_permission.find({
  permissions:{$bitsAllSet:[LOBBY_PERMISSION,CAMPUS_PERMISSION]}
});
```

This line runs a query on the `students_permission` collection in the database. The query uses the `$bitsAllSet` operator to find documents where both the `LOBBY_PERMISSION` and `CAMPUS_PERMISSION` bits are set in the `permissions` field.

Explanation of \$bitsAllSet Operator:

- The `$bitsAllSet` operator checks if all of the specified bit positions in a binary representation of a number are set to 1.
- In this example, `LOBBY_PERMISSION` (1) and `CAMPUS_PERMISSION` (2) correspond to bit positions 0 and 1, respectively.

- The `permissions` field value is checked to ensure both of these bits are set.

GEOSPATIAL:

To perform geospatial queries in MongoDB, you need to use geospatial indexes and queries. MongoDB supports a variety of geospatial queries, including finding documents near a specific point, within a certain distance, or within a polygon. Here's a step-by-step guide on how to perform these queries:

1. Insert Geospatial Data

First, insert some documents with geospatial data. Geospatial data is typically stored in the GeoJSON format.

To perform geospatial queries, you need to create a geospatial index on the field containing the geospatial data. Here's how to create a 2dsphere index for GeoJSON data:

`_id : 1`

`name : "Coffee Shop A"`

`location : Object`


`type : "Point"`

`coordinates : Array(2)`

GEOSPATIAL QUERY:

In MongoDB, a geospatial query is used to retrieve documents based on their geographical location. These queries utilize special operators like \$geoNear, \$geoWithin, and \$near to find

```
db> db.locations.find({
...   location:{
...   $geoWithin:{
...   $centerSphere:[[-74.005,40.712],0.00621376]
...   }
...   }
... });
[
  {
    _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 ] }
  }
]
db>
```



\$geoWithin: This operator selects documents with geospatial data within a specified geometry.

\$centerSphere: This operator specifies a circle for a geospatial query. The circle is defined by a center point and a radius measured in radians.

- `[[-74.005, 40.712], 0.00621376]`: This array specifies the center of the circle and its radius.
- `[-74.005, 40.712]` : This array specifies the coordinates of the center point (longitude, latitude).
- `0.00621376`: This value specifies the radius of the circle in radians. The radius is calculated based on the Earth's radius in the specified unit (usually miles or kilometers). In this case, it represents a distance of approximately 25 miles (0.00621376 radians).

Projection operations

Projection operators in MongoDB are used to shape the documents returned in the query results by including, excluding, or manipulating fields. Here are some commonly used projection operators.

1. Include Specific Fields: -

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

2. Exclude Specific Fields:

```
db.collection.find({}, { field1: 0, field2: 0 })
```

The examples for using projection operators in MongoDB is described below:

1.including the specific fields

```
{
  "_id": 1,
  "name": "Alice", "age":
  25,
  "email": "alice@example.com"
}
```

Include only the "name" and "email" fields, excluding "age" `db.users.find({}, { name: 1, email: 1, age: 0 });`

```
{ "name": "Alice", "email":
"alice@example.com" }
```

RETRIVE NAME,AGE AND GPA:

```
db> db.candidates.find({}, {name:1,age:1,gpa:1});
[
  {
    _id: ObjectId('666490b3f378263ae82e00da'),
    name: 'Alice Smith',
    age: 20,
    gpa: 3.4
  },
  {
    _id: ObjectId('666490b3f378263ae82e00db'),
    name: 'Bob Johnson',
    age: 22,
    gpa: 3.8
  },
  {
    _id: ObjectId('666490b3f378263ae82e00dc'),
    name: 'Charlie Lee',
    age: 19,
    gpa: 3.2
  },
  {
    _id: ObjectId('666490b3f378263ae82e00dd'),
    name: 'Emily Jones',
    age: 21,
    gpa: 3.6
  },
  {
    _id: ObjectId('666490b3f378263ae82e00de'),
    name: 'David Williams',
    age: 23,
    gpa: 3
  },
  {
    _id: ObjectId('666490b3f378263ae82e00df'),
    name: 'Fatima Brown',
    age: 18,
    gpa: 3.5
  }
]
```

In this MongoDB query, `db.candidates.find({}, {name:1,age:1,gpa:1})`, the `find` method is used to retrieve documents from the `candidates` collection. The first argument, `{}`, is an empty query filter, meaning all documents in the collection are selected. The second argument, `{name:1, age:1, gpa:1}`, is a projection that specifies only the `name`, `age`, and `gpa` fields should be included in the output, while excluding other fields. The result is a list of documents where each document contains only the `_id`, `name`, `age`, and `gpa` fields. The output includes details of candidates such as Alice Smith, Bob Johnson, and Charlie Lee, displaying their names, ages, and GPAs, providing a concise view of this specific subset of data from the collection.

VARIATION:EXCLUDE FIELDS:

```
db> db.candidates.find({}, {_id:0,courses:0});
[
  {
    name: 'Alice Smith',
    age: 20,
    gpa: 3.4,
    home_city: 'New York City',
    blood_group: 'A+',
    is_hotel_resident: true
  },
  {
    name: 'Bob Johnson',
    age: 22,
    gpa: 3.8,
    home_city: 'Los Angeles',
    blood_group: 'O-',
    is_hotel_resident: false
  },
  {
    name: 'Charlie Lee',
    age: 19,
    gpa: 3.2,
    home_city: 'Chicago',
    blood_group: 'B+',
    is_hotel_resident: true
  },
  {
    name: 'Emily Jones',
    age: 21,
    gpa: 3.6,
    home_city: 'Houston',
    blood_group: 'AB-',
    is_hotel_resident: false
  }
]
```

\$elemMatch:

In MongoDB, \$elemMatch is an operator used to match documents that contain an array field with at least one element that matches all the specified query criteria. It is commonly used in two scenarios: within the find query to filter documents and within projection to specify which array elements should be included in the returned documents.

```
db> db.players.find({}, {games: {$elemMatch: {score: {$gt: 5}}}, joined: 1, lastLogin: 1})
[
  {
    _id: ObjectId('60bf1ad4366e071b0405a8f8'),
    joined: '2020-01-01',
    lastLogin: '2024-06-07',
    games: [ { game: 'game2', score: 6 } ]
  },
  {
    _id: ObjectId('60bf1ad4366e071b0405a8f9'),
    joined: '2021-02-15',
    lastLogin: '2024-06-06',
    games: [ { game: 'game2', score: 8 } ]
  }
]
db>
```

This MongoDB query searches the `players` collection for documents where the `games` array contains at least one subdocument where the `score` is greater than 5. For each matching document, it projects the `joined` and `lastLogin` fields along with the `games` array containing only the subdocument that matches the condition (`score` greater than 5).

In the result, you have two documents that match the query:

1. Player1 (`_id`: '60bf1ad4366e071b0405a8f8'): This player joined on '2020-01-01' and last logged in on '2024-06-07'. They have one game (`game2`) with a score of 6.
2. Player2 (`_id`: '60bf1ad4366e071b0405a8f9'): This player joined on '2021-02-15' and last logged in on '2024-06-06'. They have one game (`game2`) with a score of 8.


\$slice:

In MongoDB, the \$slice operator is used within a projection to limit the number of elements returned from an array field. This is useful when you want to retrieve only a subset of an array from a document.

Syntax:

```
db.collection.find(  
  
    <query>,  
  
    { <arrayField>: { $slice: <number> } }  
  
);
```

```
db> db.candidates.find({courses:{$elemMatch:{$eq:"Physics"}}},{name:1,"courses,$":1});  
[  
  { _id: ObjectId('6657ff95946a866dbb971e60'), name: 'Bob Johnson' },  
  { _id: ObjectId('6657ff95946a866dbb971e62'), name: 'Emily Jones' }  
]  
db> ■
```



courses from their `courses` array. For example:

- Alice Smith is taking 'English' and 'Biology'.
- Bob Johnson is studying 'Computer Science' and 'Mathematics'.
- Charlie Lee is enrolled in 'History' and 'English'.
- Emily Jones is pursuing 'Mathematics' and 'Physics'.
- David Williams is studying 'English' and 'Literature'.
- Fatima Brown is taking 'Biology' and 'Chemistry'.
- Gabriel Miller is enrolled in 'Computer Science' and 'Engineering'.
- Hannah Garcia is studying 'History' and 'Political Science'.
- Isaac Clark is pursuing 'English' and 'Creative Writing'.
- Jessica Moore is studying 'Biology' and 'Ecology'.
- Kevin Lewis is taking 'Computer Science' and 'Artificial Intelligence'.
- Lily Robinson is enrolled in 'History' and 'Art History'.

The `\$slice` operator helps to limit the number of elements returned in an array projection, useful for scenarios where you want to retrieve a subset of elements from an array field.

CLASS - 6

AGGREGATION OPERATORS

Aggregation operations process multiple documents and return computed results. You can use aggregation operations to:

- Group values from multiple documents together.
- Perform operations on the grouped data to return a single result.
- Analyse data changes over time.

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" }



\$group

The \$group stage is used to group documents based on one or more fields and perform aggregation operations on the grouped data. It allows you to:

- Group documents by one or more fields
- Perform aggregation operations on the grouped data, such as sum, average, count, etc.
- Create new fields that represent the aggregated values

The \$group stage takes an object as its argument, where each key is the name of a field and the value is an expression that defines the aggregation operation.

\$project

The \$project stage is used to transform and reshape the data in the pipeline. It allows you to:

- Add new fields to the documents
- Rename existing fields
- Remove fields
- Perform calculations and transformations on fields
- Create new arrays or objects

The \$project stage takes an object as its argument, where each key is the name of a field and the value is an expression that defines the transformation.

Average GPA of All Students:

The average GPA of all students in MongoDB is calculated using an aggregation query that groups all documents in the `students` collection and computes the mean of the `gpa` field. This is achieved by using the `\$avg` operator within a `\$group` stage, setting `_id` to `null` to consider the entire collection.

```
test> use db
switched to db db
db> db.students.aggregate([
...   {$group: {_id: null, averageGPA: {$avg: "$gpa"}}}
...   ]);
[ { _id: null, averageGPA: 3.2268699186991867 } ]
db> |
```

`students` collection. By using the `db.students.aggregate` method, it executes an aggregation pipeline with a single `\$group` stage. In this stage, setting `_id` to `null` groups all documents

the average value of the `gpa` field across all documents using the `\$avg` operator. The output is a single document with `_id: null` (indicating no specific grouping) and `averageGPA: 3.2268699186991867`, representing the average GPA of the students.

Minimum and Maximum Age:

The minimum and maximum age in MongoDB can be calculated using an aggregation query that groups all documents in a collection and computes the minimum and maximum values of the `age` field. This is achieved by using the `\$min` and `\$max` operators within a `\$group` stage, setting `_id` to `null` to consider the entire collection.

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

students in the `students` collection. Using the `db.students.aggregate` method, it runs an

documents together. Within this group, `minAge` is calculated using the `\$min` operator to find the lowest `age` value, and `maxAge` is calculated using the `\$max` operator to find the highest `age` value. The output is a single document `{ _id: null, minAge: 18, maxAge: 25 }`, showing the minimum age as 18 and the maximum age as 25 among all students.

To find the Minimum and maximum age:

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

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.

To find the Minimum and maximum GPA:

```
db> db.students.aggregate([{$group: {_id: null, maxGpa: {$max: "$gpa"}}}]);
[ { _id: null, maxGpa: 3.99 } ]
db> db.students.aggregate([{$group: {_id: null, minGpa: {$min: "$gpa"}}}]);
[ { _id: null, minGpa: 2.51 } ]
```

minGpa: Uses the \$min operator to find the minimum value in the "gpa" field.

maxGpa: Uses the \$max operator to find the maximum value in the "gpa" field.

To get average GPA for all the home cities

```
{ _id: 'City 10', averageGPA: 3.24925 }
db> db.students.aggregate([{$group: {_id: "$home_city", averageGPA: {$avg: "$gpa"}}}]);
[
  { _id: null, averageGPA: 3.3206474820143885 },
  { _id: 'City 7', averageGPA: 3.2042857142857137 },
  { _id: 'City 5', averageGPA: 3.366470588235294 },
  { _id: 'City 9', averageGPA: 3.381111111111111 },
  { _id: 'City 1', averageGPA: 3.3738709677419356 },
  { _id: 'City 6', averageGPA: 3.239375 },
  { _id: 'City 3', averageGPA: 3.3045161290322578 },
  { _id: 'City 2', averageGPA: 3.2856666666666663 },
  { _id: 'City 8', averageGPA: 3.3918518518518517 },
  { _id: 'City 4', averageGPA: 3.1856 },
  { _id: 'City 10', averageGPA: 3.24925 }
]
```

Collect Unique Courses Offered (Using \$addToSet):

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


AGGREGATION PIPELINES

An aggregation pipeline is a sequence of stages that process data from a MongoDB collection. Each stage performs a specific operation on the data, such as filtering, transforming, or aggregating. The output of one stage becomes the input for the next stage, allowing you to chain multiple operations together to achieve complex data processing tasks.

Components of an Aggregation Pipeline

An aggregation pipeline consists of the following components:

1. **Stages:** These are the individual operations that make up the pipeline. MongoDB provides a range of built-in stages, such as `$match`, `$project`, `$filter`, `$group`, and more.
2. **Operators:** These are the specific functions or expressions used within a stage to perform the desired operation. For example, the `$match` stage uses the `$eq` operator to filter documents based on a specific condition.
3. **Expressions:** These are the values or calculations used as input to an operator. For example, in the `$project` stage, you might use an expression like `{ $concat: ["$firstName", " ", "$lastName"] }` to create a new field.

Types of Aggregation Stages

MongoDB provides several types of aggregation stages, including:

1. **Filter Stages:** These stages filter out documents that don't match a specific condition. Examples include `$match` and `$filter`.
2. **Transformation Stages:** These stages transform or modify the data in some way. Examples include `$project`, `$addFields`, and `$replaceRoot`.

Sample Data

Assume we have a **students** collection with documents like this:

```
{
  "name": "John Doe",
  "age": 21,
  "major": "Computer Science",
  "gpa": 3.5
},
{
  "name": "Jane Smith",
  "age": 22,
  "major": "Mathematics",
  "gpa": 3.8
},
{
  "name": "Alice Johnson",
  "age": 23,
```

Explanation of Operators:

- `$match` : Filters documents based on a condition.
- `$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.

These queries demonstrate various aggregation operations using the `students6` collection. Feel free to experiment with different conditions and operators to explore the power of aggregation pipelines in MongoDB.

Example 04

Find students with age greater than 23, sorted by age in descending order, and only return name and age

```
db.students6.aggregate([
  { $match: { age: { $gt: 23 } } }, // Filter students older than 23
  { $sort: { age: -1 } }, // Sort by age descending
  { $project: { _id: 0, name: 1, age: 1 } } // Project only name and
])
```

Solution

```
db> db.students6.aggregate([
...   { $match: { age: { $gt: 23 } } }, // Filter students older than 23
...   { $sort: { age: -1 } }, // Sort by age descending
...   { $project: { _id: 0, name: 1, age: 1 } } // Project only name and age
... ])
[ { name: 'Charlie', age: 28 }, { name: 'Alice', age: 25 } ]
db>
```

Example 05

Find students with age less than 23, sorted by name in ascending order, and only return name and score

```
db> db.std6.aggregate([{$match :{age:{$gt:23}}},{ $sort:{age:1}},{ $project:{_id:0,name:1,age:1}}])
[ { name: 'Alice', age: 25 }, { name: 'Charlie', age: 28 } ]
```

Example 06

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 }
]
```

Example 07

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

```
db> db.std6.aggregate([{$project:{_id:0,name:1,averageScore:{ $avg: "$scores" }}},{ $match:{averageScore:{$gt:85}}},{ $skip:1}])
[ { name: 'Alice', averageScore: 93.33333333333333 } ]
```

```
db>
Find students with age greater than 23, sorted by age in descending order, and only return name and age
```

Example:

1. Finding students with ages greater than 23, sorted by age in descending order, and only returning name and age.

```
db> db.stu6.aggregate([
...   {$match: {age: { $gt:23}}},
...   {$sort:{age:-1}},
...   {$project:{_id:0,name:1, age:1}}])
[ { name: 'Charlie', age: 28 }, { name: 'Alice', age: 25 } ]
db> |
```

2. Finding students with ages lesser than 23, sorted by name in ascending order, and only returning names and scores.

```
db> db.stu6.aggregate([ { $match: { age: { $lt: 23 } } }, {  
  $sort: { name: 1 } }, { $project: { _id: 0, name: 1, scores:  
    1 } } ] )  
[  
  { name: 'Bob', scores: [ 90, 88, 95 ] },  
  { name: 'David', scores: [ 98, 95, 87 ] }  
]  
db> |
```

3. Grouping the students by major & calculating their average age with a total number of students in each major.

```
db> db.stu6.aggregate([  
  ... { $group: { _id: "$major", avgAge: { $avg: "$age" }, totalStu: { $  
    sum: 1 } } } ] )  
[  
  { _id: 'English', avgAge: 28, totalStu: 1 },  
  { _id: 'Mathematics', avgAge: 22, totalStu: 1 },  
  { _id: 'Computer Science', avgAge: 22.5, totalStu: 2 },  
  { _id: 'Biology', avgAge: 23, totalStu: 1 }  
]  
db> |
```

ACID Properties in MongoDB

ACID stands for Atomicity, Consistency, Isolation, and Durability. These properties ensure the reliable processing of database transactions. MongoDB provides ACID properties at the document level.

Atomicity:

It ensures that a series of operations within a single transaction are completed entirely or not at all.

In MongoDB, write operations on a single document (inserts, updates, deletes) are atomic, which means they either fully succeed or fully fail, that is, atomicity guarantees that all of the commands that make up a transaction are treated as a single unit and either succeed or fail together.

This is important as in the case of an unwanted event, like a crash or power outage, we can be sure of the state of the database. The transaction would have either been completed successfully or rolled back if any part of the transaction failed..

Example:

javascript

```
db.collection.insertOne({  
  _id: 1,  
  name: "Alice",  
  balance: 500  
});
```

-

Consistency

- In MongoDB, consistency is maintained through replica sets. When a write operation is performed, it is first applied to the primary node and then replicated to the secondary nodes.
- Read operations can be configured to read from the primary node (ensuring strong consistency) or from secondary nodes (eventual consistency). MongoDB provides read concerns that allow developers to control the level of consistency for their read operations.

Isolation

- MongoDB ensures isolation at the document level. This means that during a write operation, the document being modified is locked, preventing other operations from reading or writing the document until the operation is complete.
- For multi-document transactions, MongoDB uses snapshot isolation, ensuring that transactions provide a consistent view of the data and are isolated from other operations.

Durability

- Durability in MongoDB is achieved through journaling. Write operations are written to the journal before being applied to the data files. This ensures that in the event of a crash, the operations can be recovered from the journal.
- Replica sets further enhance durability by replicating data across multiple nodes, ensuring that data is not lost even if a node fails.

Replication

Replication involves copying data from one MongoDB server (primary) to one or more MongoDB servers (secondary). It provides data redundancy and increases data availability.

Features of Replication:

1. Data Redundancy:

- Multiple copies of the data ensure that data is not lost if a server fails.

2. High Availability:

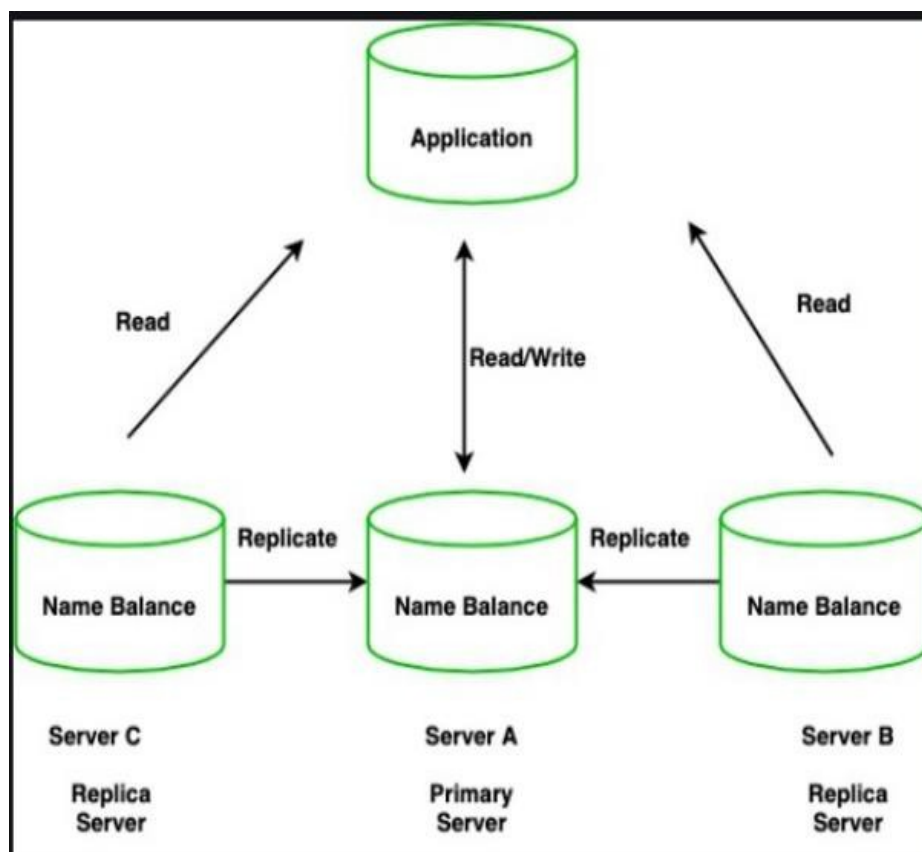
- If the primary server fails, one of the secondary servers can be promoted to the primary, ensuring the database remains available.

3. Automatic Failover:

- In a replica set, if the primary server goes down, the secondary servers hold an election to choose a new primary automatically.

4. Read Scaling:

- Read operations can be distributed across multiple secondaries to improve read performance.



Sharding

Sharding is a method for distributing data across multiple machines. It is used to support deployments with very large data sets and high throughput operations.

Features of Sharding:

1. Horizontal Scaling:

- Data is distributed across multiple servers, each holding a subset of the data (shards).

2. Data Distribution:

- Sharding allows MongoDB to distribute data across multiple servers, balancing the load and improving performance.

3. Query Routing:

- MongoDB automatically routes queries to the appropriate shard(s) based on the data distribution.

4. Large Data Set Handling:

- Enables MongoDB to handle very large datasets that exceed the capacity of a single server.

Comparison of Replication and Sharding

Feature	Replication	Sharding
Purpose	Data redundancy and high availability	Horizontal scaling and data distribution
Data Distribution	Copies of the same data on multiple servers	Different subsets of data on different servers
Failover	Automatic failover with election of new primary	No automatic failover; designed for scaling
Read Scalability	Yes, read operations can be distributed	Yes, but primarily for distributing data load
Write Scalability	No, all writes go to the primary	Yes, writes are distributed across shards
Setup Complexity	Moderate, involves setting up replica sets	High, involves setting up shards, config servers, and routers

Indexes in MongoDB

Indexes in MongoDB are special data structures that store a small portion of the data set in an easy-to-traverse form. They support the efficient execution of queries by providing fast access to documents. Here's a detailed overview of indexes in MongoDB:

Types of Indexes

1. Single Field Index:

- An index on a single field.
- Example: `db.collection.createIndex({ field: 1 })` for ascending order or `db.collection.createIndex({ field: -1 })` for descending order.

2. Compound Index:

- An index on multiple fields.
- Example: `db.collection.createIndex({ field1: 1, field2: -1 })`.

3. Multikey Index:

- An index on an array field where MongoDB creates an index for each element in the array.
- Example: `db.collection.createIndex({ arrayField: 1 })`.

4. Text Index:

- An index to support text search for string content.
- Example: `db.collection.createIndex({ field: "text" })`.

5. Hashed Index:

- An index that hashes the value of a field to support hash-based sharding.
- Example: `db.collection.createIndex({ field: "hashed" })`.

6. Geospatial Indexes:

- **2dsphere Index:** Supports queries for spherical geometry on a globe.
 - Example: `db.collection.createIndex({ location: "2dsphere" })`.
- **2d Index:** Supports queries for planar geometry on a flat surface.
 - Example: `db.collection.createIndex({ location: "2d" })`.

7. Wildcard Index:

- Indexes all fields and subfields in a document.
- Example: `db.collection.createIndex({ "$**": 1 })`.

Additional Considerations

- **Sparse Indexes:**
 - Only index documents where the indexed field exists.
 - Can improve performance for sparse datasets.
- **Unique Indexes:**
 - Ensure that the indexed field has unique values across all documents.
- **TTL Indexes:**
 - Automatically expire documents after a specified time.

Choosing the right index type depends on your specific data structure, query patterns, and performance requirements. Careful index design can significantly improve query performance, but excessive indexing can impact write performance.

Would you like to delve deeper into a specific index type or discuss index creation strategies for a particular use case?