MONGODB

Projection

Projection in MongoDB means selecting only certain fields from a document in a query. This helps to retrieve only the necessary data and improve query speed. Projection is handy when you only require a specific set of fields from a document, not the entire document.

When querying a MongoDB collection, you can specify a projection to limit the amount of data returned by the query.

Syntax of Projection Operators

- **Include Fields**

  Use the projection document {field1: 1, field2: 1} to only return the specified fields.

  (1- True, 0-False)

- **Exclude Fields**

  Use the projection document {field1: 0, field2: 0} to exclude the specified fields.

- **Mixed Inclusion/Exclusion**

  You can't mix inclusion and exclusion in the same projection document

It alloes you to select only the necessary data rather than selecting whole dta from the document.

For example, a document contains 5 fields i.e,

```
{
name: "ved",
age: 20,
branch: EEE,
usn:4AI22CD001,
blood_group:”A+”
}
```

In the below example we are finding in student database that should get the details of both name and age , in the output will be getting the name ,age with the default ‘_id’ .

```
db> db.student.find({},{name:1,age:1});
[
  {
    _id: ObjectId('665de5dd6e26f71bef17d066'),
    name: 'Student 948',
    age: 19
  },
  {
    _id: ObjectId('665de5dd6e26f71bef17d067'),
    name: 'Student 157',
    age: 20
  },
  {
    _id: ObjectId('665de5dd6e26f71bef17d068'),
    name: 'Student 316',
    age: 20
  },
  {
    _id: ObjectId('665de5dd6e26f71bef17d069'),
    name: 'Student 346',
    age: 25
  },
  {
```

To exclude the "_id" we will just modify the syntax by add a new condition saying that "_id:0" (i.e, it won't print the details of the id)

```
db> db.student.find({},{name:1,age:1,_id:0});
[
  { name: 'Student 948', age: 19 },
  { name: 'Student 157', age: 20 },
  { name: 'Student 316', age: 20 },
  { name: 'Student 346', age: 25 },
  { name: 'Student 930', age: 25 },
  { name: 'Student 305', age: 24 },
  { name: 'Student 268', age: 21 },
  { name: 'Student 563', age: 18 },
  { name: 'Student 440', age: 21 },
  { name: 'Student 536', age: 20 },
  { name: 'Student 256', age: 19 },
  { name: 'Student 177', age: 23 },
  { name: 'Student 871', age: 22 },
```

Projecting Specific Fields

- Top-Level Fields

  Project top-level fields like {name: 1, age: 1} to return only those fields.

- Nested Fields

  Project nested fields like {"address.city": 1, "address.state": 1} to target specific subfields.

- Array Elements

  Project array elements like {friends: 1} to return the full array.

```
db.users.insertOne({
  "name": "Jane Smith",
  "age": 28,
  "occupation": "Designer",
  "address": {
    "street": "456 Elm St",
    "city": "Othertown",
    "state": "NY",
    "zip": "67890"
  }
})
```

Nested Fields

To project nested fields, you can specify the field names with dot notation, like this:

```
db> db.users.find({}, { "address.city": 1, "address.state": 1 })

[
  { _id: ObjectId('6665cef44a907c5d2fcdce00') },
  {
    _id: ObjectId('66681a790dd50348cacdcdf6'),
    address: { city: 'Anytown', state: 'CA' }
  },
  {
    _id: ObjectId('66681ae50dd50348cacdcdf7'),
    address: { city: 'capetown', state: 'xi' }
  },
  {
    _id: ObjectId('66681b120dd50348cacdcdf8'),
    address: { city: 'Othertown', state: 'NY' }
  },
  {
    _id: ObjectId('66681b1a0dd50348cacdcdf9'),
    address: { city: 'Thistown', state: 'TX' }
  },
  {
    _id: ObjectId('66681b230dd50348cacdcdfa'),
    address: { city: 'Thatstown', state: 'FL' }
```

Ignore Attributes

```
db> db.users.find({}, {_id:0})
[
  { name: 'nivi', age: 21 },
  {
    name: 'John Doe',
    age: 30,
    ' occupation': 'Developer',
    address: {
      street: '123 Main St',
      city: 'Anytown',
      state: 'CA',
      zip: '12345'
    }
  },
  {
    name: 'xyz',
    age: 25,
    ' occupation': 'Developer',
    address: { street: '234St', city: 'capetown', state: 'xi', zip: '12356' }
  },
  {
    name: 'Jane Smith',
    age: 28,
    occupation: 'Designer',
    address: {
      street: '456 Elm St',
      city: 'Othertown',
      state: 'NY',
      zip: '67890'
    }
  },
  {
```

LIMIT

The $limit Operator

- Purpose

  The $limit operator in MongoDB is used to restrict the number of documents returned by query

- Syntax

  The $limit operator takes a single argument, which is the maximum number of documents to return.

- Example

  db.collection.find().limit(5) // Returns up to 5 documents

Syntax and Usage

- Limiting Queries

  The $limit operator can be used in various MongoDB queries, such as find(), aggregate(), and count().

- Chaining Operators

  $limit can be combined with other query operators like $skip, $sort, and $project to create complex data retrieval pipelines.

- Negative Limits

  Passing a negative value to $limit will return the last n documents, which can be useful for retrieving the "top" or "bottom" records.

- **Syntax**

```
db.COLLECTION_NAME.find().limit(NUMBER)
// Here, COLLECTION_NAME is the name of the collection you want to query, and NUMBER is the number of documents you want to limit the result to.
```

```
db> db.users.find().limit(-5)
[
  { _id: ObjectId('6665cef44a907c5d2fcdce00'), name: 'nivi', age: 21 },
  {
    _id: ObjectId('66681a790dd50348cacdcdf6'),
    name: 'John Doe',
    age: 30,
    ' occupation': 'Developer',
    address: {
      street: '123 Main St',
      city: 'Anytown',
      state: 'CA',
      zip: '12345'
    }
  },
  {
    _id: ObjectId('66681ae50dd50348cacdcdf7'),
    name: 'xyz',
    age: 25,
    ' occupation': 'Developer',
    address: { street: '234St', city: 'capetown', state: 'xi', zip: '12356' }
  },
  {
    _id: ObjectId('66681b120dd50348cacdcdf8'),
    name: 'Jane Smith',
    age: 28,
    occupation: 'Designer',
    address: {
      street: '456 Elm St',
      city: 'Othertown',
```

Negative Limits:

This will return the last 5 documents in the collection.

This can be particularly useful when you want to retrieve the "top" or "bottom" records in a collection, such as:

Retrieving the last 5 details

```
db> db.users.find({},{age:1,name:1}).limit(-5)
[
  { _id: ObjectId('6665cef44a907c5d2fcdce00'), name: 'nivi', age: 21 },
  {
    _id: ObjectId('66681a790dd50348cacdcdf6'),
    name: 'John Doe',
    age: 30
  },
  { _id: ObjectId('66681ae50dd50348cacdcdf7'), name: 'xyz', age: 25 },
  {
    _id: ObjectId('66681b120dd50348cacdcdf8'),
    name: 'Jane Smith',
    age: 28
  },
  {
    _id: ObjectId('66681b1a0dd50348cacdcdf9'),
    name: 'Bob Johnson',
    age: 35
  }
]
db> db.users.find({},{age:1,name:1,_id:0}).limit(-5)
[
  { name: 'nivi', age: 21 },
  { name: 'John Doe', age: 30 },
  { name: 'xyz', age: 25 },
  { name: 'Jane Smith', age: 28 },
  { name: 'Bob Johnson', age: 35 }
]
```

Combining $limit with Other Aggregation Stages

Filtering and Sorting

$limit can be used alongside $match and $sort stages to retrieve the top or bottom records that match specific criteria.

Data Transformation

$limit can be combined with stages like $project and $group to limit the output of complex data transformation pipelines.

Performance Optimization

Judiciously using $limit can improve the performance of your aggregation pipeline by reducing the amount of data that needs to be processed.

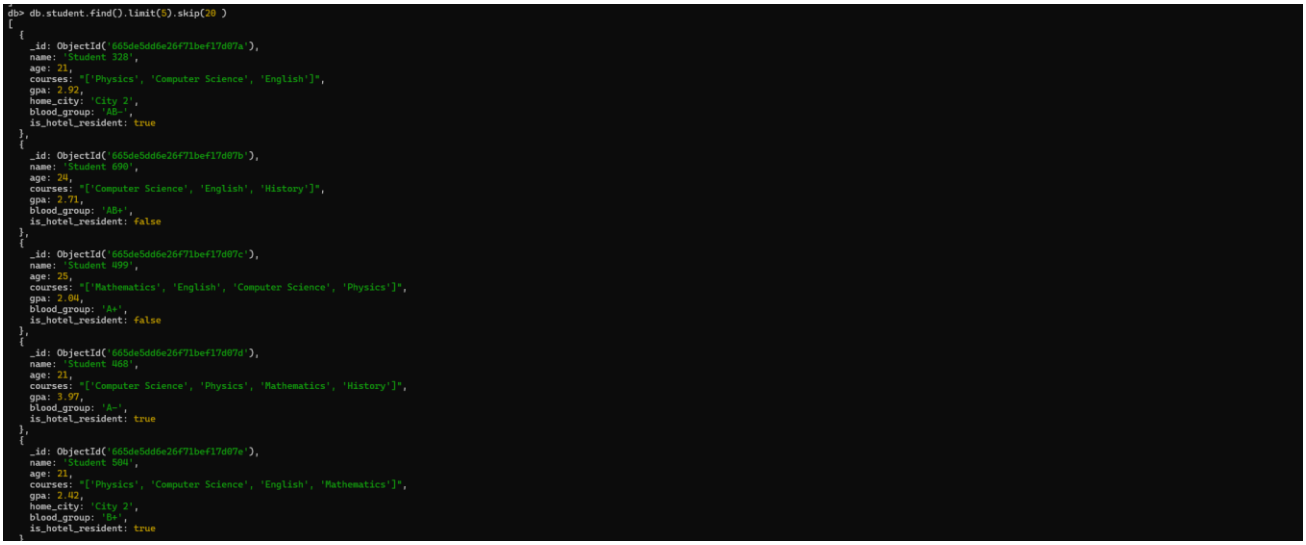
Pagination with $limit and $skip

1. db.users.find().limit(5).skip(0)
   - db.users.find(): This is the basic query to retrieve documents from the users collection.
   - limit(5): This operator limits the number of documents returned to 5. It's like saying "give me only 5 documents".
   - skip(0): This operator skips a specified number of documents before returning the results. In this case, it's skipping 0 documents, which means it will start returning documents from the beginning of the collection.

2. db.users.find().limit(5).skip(10)
   - find(): Retrieves all documents from the users collection.
   - limit(5): Limits the number of documents returned to 5.
   - skip(10): Skips the first 10 documents and returns the next 5 documents.

3. db.users.find().limit(5).skip(20)

- find(): Retrieves all documents from the users collection.
- limit(5): Limits the number of documents returned to 5.
- skip(20): Skips the first 20 documents and returns the next 5 documents.



SELECTORS

In MongoDB, a selector is a part of a query that specifies the conditions that a document must meet to be included in the result set. It is used in the find() method to filter documents.

A selector can be a simple filter, a complex filter, or a combination of filters. Here are some examples of selectors:

Simple Selectors

- Equality: **db.collection.find({ name: {$eq:"John"}})**

  - finds documents where the **name** field is "John"

- Inequality: **db.collection.find({ age: { $ne: 30 } })**

  - finds documents where the **age** field is not 30

- Greater than: **db.collection.find({ age: { $gt: 30 } })**

  - finds documents where the **age** field is greater than 30

- Less than: **db.collection.find({ age: { $lt: 30 } })**

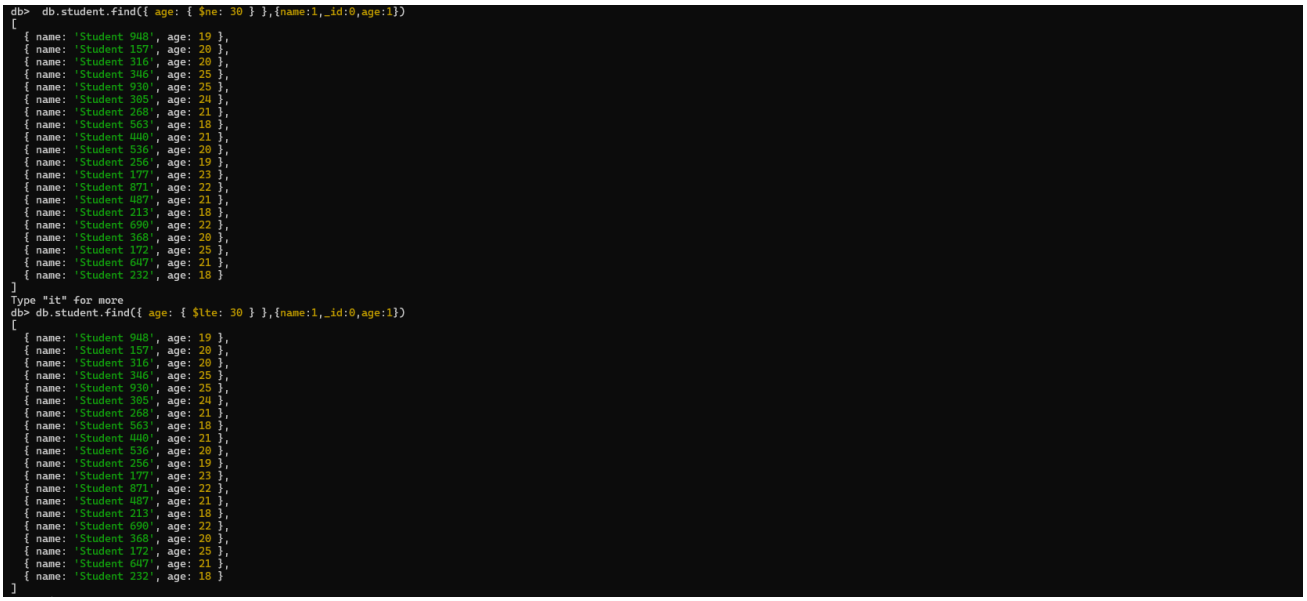
  - finds documents where the **age** field is less than 30

- Greater than or equal to: **db.collection.find({ age: { $gte: 30 } })**

  - finds documents where the **age** field is greater than or equal to 30

- Less than or equal to: **db.collection.find({ age: { $lte: 30 } })**

  - finds documents where the **age** field is less than or equal to 30



Complex Selectors

- AND: **db.collection.find({ name: "John", age: 30 })**

  – finds documents where the **name** field is "John" and the **age** field is 30

- OR: **db.collection.find({ $or: [{ name: "John" }, { age: 30 }] })**

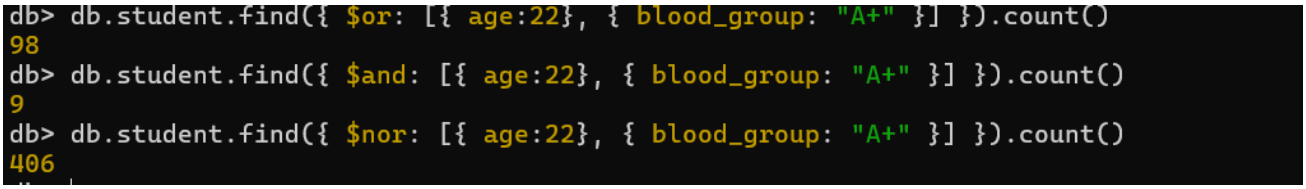
  - finds documents where the **name** field is "John" or the **age** field is 30

- NOT: **db.collection.find({ name: { $not: "John" } })**

  - finds documents where the **name** field is not "John"

- NOR: **db.collection.find({ $nor: [{ name: "John" }, { age: 30 }] }**

  - finds documents where the **name** field is not "John" and the **age** field is not 30



Array Selectors

- **db.collection.find({ scores: 80 })**

  - finds documents where the **scores** array contains the value 80

- **db.collection.find({ scores: { $all: [80, 90] } })**

  - finds documents where the **scores** array contains both 80 and 90

- **db.collection.find({ scores: { $elemMatch: { $gt: 80 } } })**

  - finds documents where the **scores** array contains at least one element greater than 80

Existence Selectors

- **db.collection.find({ name: { $exists: true } })**

  - finds documents where the **name** field exists

- **db.collection.find({ name: { $exists: false } })**

  - finds documents where the **name** field does not exist

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`.

```
db> const l_p=1;

db> const c_p=2;

db> db.sp.find({
... permissions:{$bitsAllSet:[l_p,c_p]}});
[
  {
    _id: ObjectId('666852859d450f507f54827e'),
    name: 'George',
    age: 21,
    permissions: 6
  },
  {
    _id: ObjectId('666852859d450f507f54827f'),
    name: 'Henry',
    age: 27,
    permissions: 7
  },
  {
    _id: ObjectId('666852859d450f507f548280'),
    name: 'Isla',
    age: 18,
    permissions: 6
  }
]
```

Geospatial

Geospatial queries in MongoDB allow you to perform queries on geospatial data, such as locations and shapes. There are two types of geospatial indexes in MongoDB: 2d and 2dsphere.

2d Indexes 2d indexes support queries that calculate geometries on a two-dimensional plane. To create a 2d index, use the db.collection.createIndex() method, specifying the location field as the key and the string literal "2d" as the index type:

```
db.collection.createIndex( { <location field> : "2d" } )
```

2dsphere Indexes 2dsphere indexes support queries that calculate geometries on a sphere, such as the surface of the Earth. To create a 2dsphere index, use the db.collection.createIndex() method, specifying the location field as the key and the string literal **"2dsphere"** as the index type:

```
db.collection.createIndex( { <location field> : "2dsphere" } )
```

Name	Description
`$geoIntersects`	Selects geometries that intersect with a GeoJSON geometry. The `2dsphere` index supports `$geoIntersects`.
`$geoWithin`	Selects geometries within a bounding GeoJSON geometry. The `2dsphere` and `2d` indexes support `$geoWithin`.
`$near`	Returns geospatial objects in proximity to a point. Requires a geospatial index. The `2dsphere` and `2d` indexes support `$near`.
`$nearSphere`	Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The `2dsphere` and `2d` indexes support `$nearSphere`.

```
db> db.loc.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 ] }
  }
]
```

Components

- location: The field in the document that contains the geospatial data.
- $geoWithin: The query operator that specifies the geospatial query.
- $centerSphere: The specific type of geospatial query, which is a circular area centered at a point on the Earth's surface.

[-74.005, 40.712]: The center point of the circle, represented as a longitude-latitude pair. In this case, the center point is approximately New York City, USA.

0.00621376: The radius of the circle, in radians. This value corresponds to approximately 1 kilometer (0.62 miles) at the Earth's surface.