Agenda

- Pagination
- Indexes in MongoDB
- MongoDB Aggregation Pipeline

Pagination

What is pagination?

Pagination is an ordinal numbering of pages, which is usually located at the top or bottom of the site pages. In most cases, it is used for main pages ar partitions. It often looks like this:

Pagination makes users' life simpler and more convenient when it comes to distributing products on the website in a measured manner. Imagine a eCommerce website with a catalog consists of several hundred products of different categories placed on one single page. Will that confuse the use Definitely.

How pagination works in MongoDB?

- MongoDB is a document-based database so pagination is a very important use case while using MongoDB.
- When we paginate the response in MongoDB our documents result is very clear and in an understandable format.
- To paginate our MongoDB database we have using the following scenarios.

1. Process the batch file
2. To show a large amount of results set on the user screen or interface

- We cannot consider client and server-side pagination in MongoDB because it's very expensive as compared to other options.
- We are handling pagination at the database level and also we are optimizing our databases to do pagination at the database level.

In this module we are going how to simply implement pagination using **limit** and **skip** methods provided by mongodb.

The skip method will skip the specified documents from the collection while the limit method will retrieve a specified number of documents from collection.

In Mongo Shell your pagination code looks something like this

```
// Page 1
db.students.find().limit(5)

// Page 2
db.students.find().skip(5).limit(5)

// Page 3
db.students.find().skip(10).limit(5)
```

Indexes in MongoDB

Indexes are special data sets which store a partial part of the collection's data. Since the data is partial, it becomes easier to read this data. This parti set stores the value of a specific field or a set of fields ordered by the value of the field.

Indexes are very important in any database, and with MongoDB it's no different. With the use of Indexes, performing queries in MongoDB becomes mo efficient.

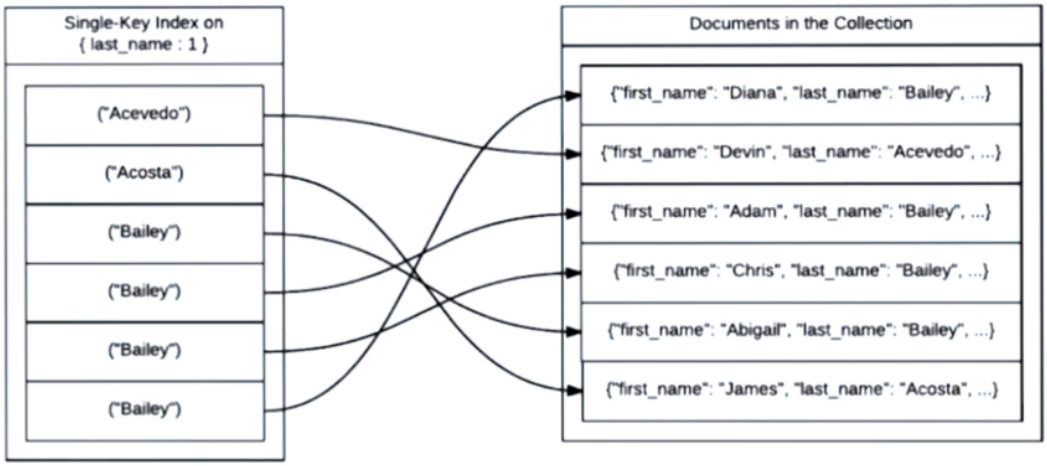
If you had a collection with thousands of documents with no indexes, and then you query to find certain documents, then in such case MongoDB wou need to scan the entire collection to find the documents. But if you had indexes, MongoDB would use these indexes to limit the number of documen that had to be searched in the collection

Collection Scan:

If you don't use an index when we query our collection, the database will have to look at every document. Our collection grows in size. Therefore we w have to search through more and more documents to satisfy our query.* Order of N operation* Linear running time

Index Scan:

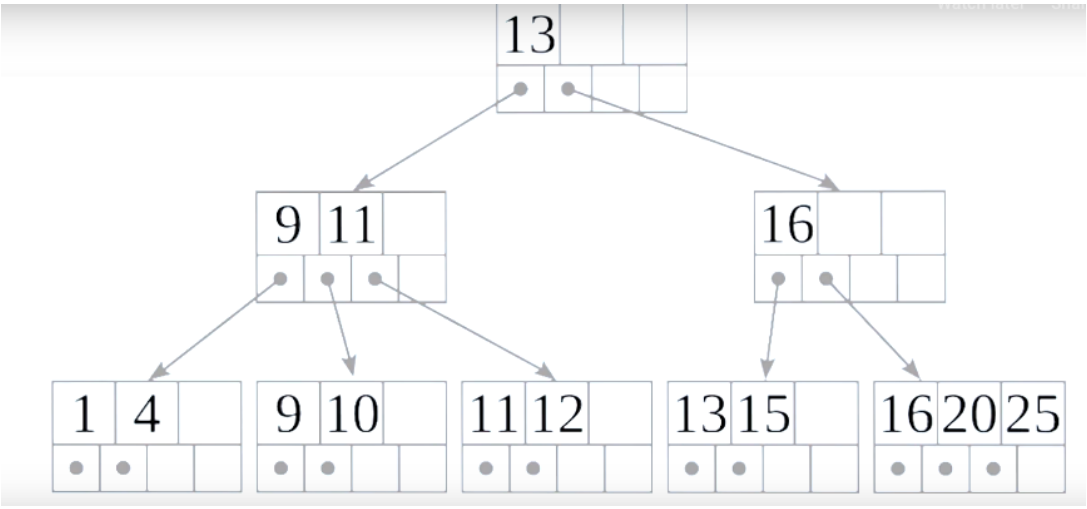
Rather than searching through every document, we can search through the ordered index first. Key-Value pair, where the key is the field's value th we've indexed on, and the key's value is the actual document itself.



> MongoDB creates a unique index on the _id field during the creation of a collection. The _id index prevents clients from inserting two documents with the same value for the _id field. You cannot drop this index on the _id field.

It is possible to have many indexes on the same collection. You might create multiple indexes in different fields if you find other queries for various fields

MongoDB uses a data structure called a b-tree to store its indexes.

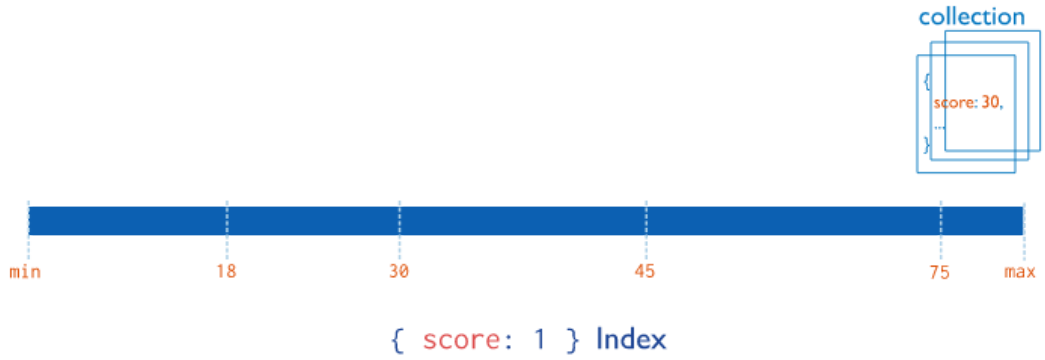


The excellent query performance gain that we get with indexes doesn't come for free. With each additional index, we decrease our write spee for a collection. If a document were to change or completely removed, one or more of our b-trees might need to be balanced. This means th we need to be careful when creating indexes. We don't want to have too many unnecessary indexes in a collection because there would b excessive loss in insert, update, and delete performance. It would help if you had a good idea of what indexes are, their pros and cons, ar how they work.

Types of Indexes

1. **Single Field Indexes:** MongoDB provides complete support for indexes on any field in a collection of documents. By default, all collections have an index on the _id field, and applications and users may add additional indexes to support important queries and operations.

This document describes ascending/descending indexes on a single field.



Create an Ascending Index on a Single Field:

Consider a collection named `records` that holds documents that resemble the following sample document:

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

The following operation creates an ascending index on the `score` field of the `records` collection:

```
db.records.createIndex( { score: 1 } )
```

The value of the field in the index specification describes the kind of index for that field. For example, a value of `1` specifies an index that orders iten in ascending order. A value of `-1` specifies an index that orders items in descending order.

The created index will support queries that select on the field `score`, such as the following:

```
db.records.find( { score: 2 } )
db.records.find( { score: { $gt: 10 } } )
```

Create an Index on an Embedded Field:

You can create indexes on fields within embedded documents, just as you can index top-level fields in documents.

Consider a collection named `records` that holds documents that resemble the following sample document:

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
  "location": { state: "NY", city: "New York" }
}
```

The following operation creates an index on the `location.state` field:

```
db.records.createIndex( { "location.state": 1 } )
```

The created index will support queries that select on the field `location.state`, such as the following:

```
db.records.find( { "location.state": "CA" } )
db.records.find( { "location.city": "Albany", "location.state": "NY" } )
```

Create an Index on Embedded Document:

You can also create indexes on embedded document as a whole.

Consider a collection named `records` that holds documents that resemble the following sample document:

```
{
  "_id": ObjectId("570c04a4ad233577f97dc459"),
  "score": 1034,
```

```
  "location": { state: "NY", city: "New York" }
}
```

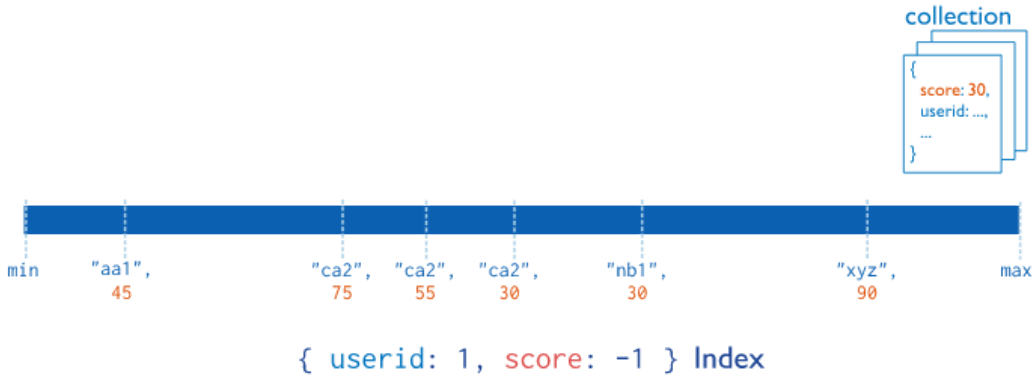
The `location` field is an embedded document, containing the embedded fields `city` and `state`. The following command creates an index c the `location` field as a whole:

```
db.records.createIndex( { location: 1 } )
```

The following query can use the index on the `location` field:

```
db.records.find( { location: { city: "New York", state: "NY" } } )
```

2. **Compound Indexes:** MongoDB supports *compound indexes*, where a single index structure holds references to multiple fields within a collection's documents. The following diagram illustrates an example of a compound index on two fields:



Compound indexes can support queries that match on multiple fields.

Create a Compound Index:

To create a compound index use an operation that resembles the following prototype:

```
db.collection.createIndex( { <field1>: <type>, <field2>: <type2>, ... } )
```

The value of the field in the index specification describes the kind of index for that field. For example, a value of `1` specifies an index that orders iten in ascending order. A value of `-1` specifies an index that orders items in descending order.

Consider a collection named `products` that holds documents that resemble the following document:

```
{
 "_id": ObjectId(...),
 "item": "Banana",
 "category": ["food", "produce", "grocery"],
 "location": "4th Street Store",
 "stock": 4,
 "type": "cases"
}
```

The following operation creates an ascending index on the `item` and `stock` fields:

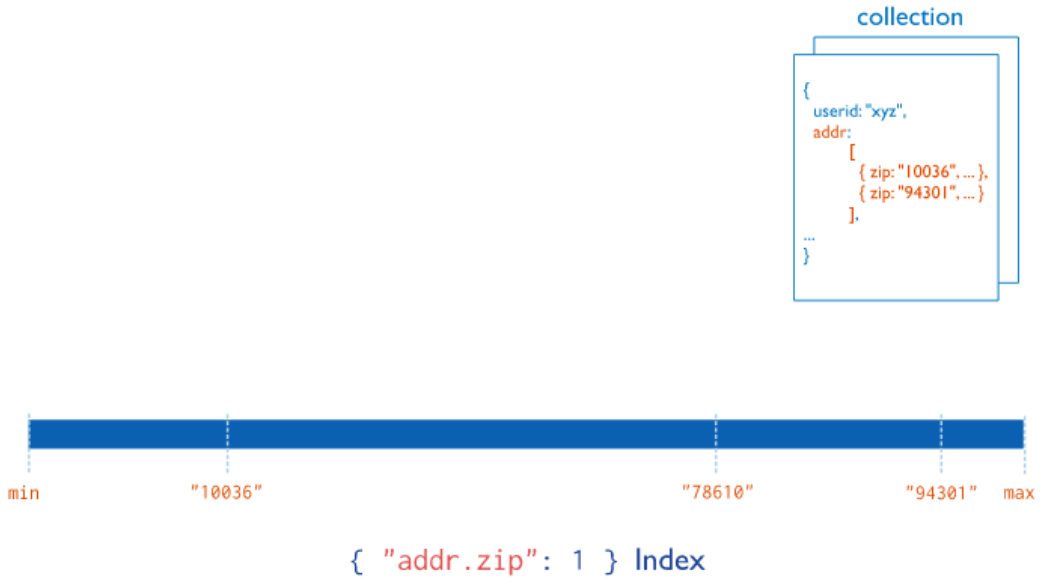
```
db.products.createIndex( { "item": 1, "stock": 1 } )
```

The order of the fields listed in a compound index is important. The index will contain references to documents sorted first by the values the `item` field and, within each value of the `item` field, sorted by values of the stock field.

In addition to supporting queries that match on all the index fields, compound indexes can support queries that match on the prefix of the index field That is, the index supports queries on the `item` field as well as both `item` and `stock` fields:

```
db.products.find( { item: "Banana" } )
db.products.find( { item: "Banana", stock: { $gt: 5 } } )
```

3. **Multikey Indexes:** To index a field that holds an array value, MongoDB creates an index key for each element in the array. These *multikey* indexes support efficient queries against array fields. Multikey indexes can be constructed over arrays that hold both scalar values (e.g. strings, numbers) *and* nested documents.



Create Multikey Index:

Suppose we have a collection called `products` that contains the following documents:

```
{ "_id" : 1, "product" : "Bat", "sizes" : [ "S", "M", "L" ] }
{ "_id" : 2, "product" : "Hat", "sizes" : [ "S", "L", "XL" ] }
{ "_id" : 3, "product" : "Cap", "sizes" : [ "M", "L" ] }
```

We can create a multikey index on that collection like this:

```
db.products.createIndex(
   {
     "sizes": 1
   }
)
```

It's just like creating a regular index. You don't need to explicitly specify that it's a multikey index. MongoDB is able to determine that the field holds a array, and therefore create it as a multikey index.

With multikey indexes, MongoDB creates an index key for each element in the array.

Compound Multikey Index on Embedded Documents:

As mentioned, you can create multikey indexes for arrays that hold embedded documents.

You can create a compound index on these, so that your index is created against multiple fields in the array.

Suppose we have a collection called `restaurants` with documents like this:

```
db.restaurants.insertMany([
   {
    _id: 1,
    name: "The Rat",
    reviews: [{
        name: "Stanley",
        date: "04 December, 2020",
        ordered: "Dinner",
        rating: 1
      },
      {
        name: "Tom",
        date: "04 October, 2020",
        ordered: "Lunch",
```

```
        rating: 2
      }]
  },
  {
   _id: 2,
   name: "Yum Palace",
   reviews: [{
       name: "Stacey",
       date: "08 December, 2020",
       ordered: "Lunch",
       rating: 3
     },
     {
       name: "Tom",
       date: "08 October, 2020",
       ordered: "Breakfast",
       rating: 4
     }]
  },
  {
   _id: 3,
   name: "Boardwalk Cafe",
   reviews: [{
       name: "Steve",
       date: "20 December, 2020",
       ordered: "Breakfast",
       rating: 5
     },
     {
       name: "Lisa",
       date: "25 October, 2020",
       ordered: "Dinner",
       rating: 5
     },
     {
       name: "Kim",
       date: "21 October, 2020",
       ordered: "Dinner",
       rating: 5
     }]
  }
])
```

We could create a compound multikey index like this:

```
db.restaurants.createIndex(
  {
    "reviews.ordered": 1,
    "reviews.rating": -1
  }
)
```

Now, the multikey index will be used whenever we run queries that involve those fields.

Here's what the query plan looks like when we search against one of those fields:

```
db.restaurants.find( { "reviews.ordered": "Dinner" } ).explain()
```

MongoDB Aggregation Pipeline

Similar to the SQL Group By clause, MongoDB can easily **batch process data** and present a single result even after executing several other operatior on the group data using MongoDB's Aggregation framework

When dealing with a database management system, any time you extract data from the database you need to execute an operation called a quer However, queries only return the data that already exists in the database. Therefore, to analyze your data to zero in on patterns or other informatic

about the data – instead of the data itself – you'll often need to perform another kind of operation called an aggregation.

MongoDB allows you to perform aggregation operations through a mechanism called MongoDB Aggregation Pipelines. These are essentially built as sequential series of declarative data operations called stages.

Each stage can then inspect and transform the documents as they pass through the pipeline, putting the transformed data results into the subseque stages for further processing. Documents from a chosen collection get into a pipeline and go through each stage, where the output coming from eac stage becomes the input for the next stage, and the final result is obtained at the end of the pipeline.

Stages can help you perform operations like:

- **Sorting**: You can reorder the documents based on a chosen field.
- **Filtering**: This operation resembles queries, where the list of documents can be narrowed down through a set of criteria.
- **Grouping**: With this operation, you can process multiple documents together to generate a summarized result.
- **Transforming**: Transforming refers to the ability to modify the structure of documents. This means you can rename or remove certain fields, or perhaps group or rename fields within an embedded document for legibility.

What are the Operators in MongoDB Aggregation Pipeline?

MongoDB provides you with an exhaustive list of operators that you can use across various aggregation stages. Each of these operators can be used construct expressions for use in the aggregation pipeline stages. Operator expressions are similar to functions that use arguments. Generally, thes expressions use an array of arguments and have the following format:

```
{ <operator> : [ <argument1>, <argument2>, ... ] }
```

However, if you only want to use an operator that accepts a single argument, you can omit the array field. It can be used in the following format:

```
{ < operator> : <argument> }
```

Here are a few different operators you can choose from:

- **Comparison Expression Operators**: This returns a boolean, except for $cmp, which will return a number.
- **Arithmetic Expression Operators**: These operators will perform mathematical operations on numbers.
- **Array Expression Operators**: With Array Expression Operators, you can perform operations on arrays.
- **Boolean Expression Operators**: These operators evaluate their argument expressions as booleans and return a boolean as a result.
- **Literal Expression Operators**: Literal Expression Operators can return a value without having to parse it first.
- **Conditional Expression Operators**: With Conditional Expression Operators, you can help build conditional statements.
- **Custom Aggregation Expression Operators**: You can use custom aggregation expression operators to define custom aggregation functions.
- **Object Expression Operators**: These allow you to merge or split documents.
- **Date Expression Operators**: Date expression operators returns date components or objects of a given date object.
- **Text Expression Operators**: These operators allow you to access per-document metadata per aggregation.
- **String Expression Operators**: With the help of these operators, you can perform well-defined behavior for strings of ASCII characters.
- **Trigonometry Expression Operators**: These operators can perform trigonometric operations on numbers.
- **Type Expression Operators**: You can use these operators to perform operations on the data type.
- **Variable Expression Operators**: These operators can define variables for use within the scope of a subexpression and return the result of that subexpression.

7 Key MongoDB Aggregation Pipeline Stages

Every stage of the MongoDB Aggregation Pipeline transforms the document as the documents pass through it. However, once an input document passe through a stage, it doesn't necessarily produce one output document. Some stages might create more than one document as a result.

MongoDB offers its users the db.collection.aggregate() method in the mongo shell along with the db.aggregate() command to run the aggregatic pipeline. A stage can show up multiple times within a pipeline, with the exception of $merge, $out, and $geoNear stages.

- $match
- $group
- $project
- $sort
- $skip

- $limit
- $unwind

$match

This MongoDB Aggregation Pipeline stage filters the document stream to allow only matching documents to pass unmodified into the next pipeline stag For every input document, the output is either zero document (no match) or one document (a match).

$group

With this MongoDB Aggregation Pipeline stage, you can group input documents by a specified identifier expression and apply the accumulator(expressions, if mentioned, to every group. $group ends up consuming all input documents and gives one document per each distinct group. The outp documents will only contain the identifier fields, and if mentioned, the accumulated fields.

$project

This MongoDB Aggregation Pipeline stage can reshape every document in the stream, for instance, by adding new fields or getting rid of existing field For every input document, you can provide one document as an output.

$sort

With $sort, you can reorder the document streams with a specified sort key. The documents are unmodified, leave for the order of the documents. F every input document, the output for this MongoDB Aggregation Pipeline stage is a single document.

$skip

$skip allows you to skip the first n documents where n is the specified skip number and passes the remaining documents unamended to the pipeline. F every input document, the output for this MongoDB Aggregation Pipeline stage is either a zero document (after the first n documents) or one docume (for the first n documents).

$limit

This MongoDB Aggregation Pipeline stage allows you to pass the first n documents unamended to the pipeline where n is the specified limit. For eve input document, the output is either a zero document (after the first n documents) or one document (for the first n documents).

$unwind

This MongoDB Aggregation Pipeline can break an array field from the input documents and outputs one document for every element. Every outp document will contain the same field, but the array field gets replaced by an element value per document. For every input document, $unwind will outp n documents where n is the number of elements and could even be zero for an empty array.

For more information on MongoDB Aggregation Pipeline stages, you can give MongoDB Aggregation Pipeline Stages a read.

How to Run the MongoDB Aggregation Pipeline?

Step 1: Setting up the Connection

First, you need to open a connection to the database by using the command ‘mongo’. When you see ‘>’ within the prompt then you are ready to perfor the commands related to the database operations.

```
C:\Program Files\MongoDB\Server\4.4\bin>mongo
MongoDB shell version v4.4.4
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("04d2f324-6e2f-4efe-bfcc-9e33575718f3") }
MongoDB server version: 4.4.4
---
The server generated these startup warnings when booting:
        2021-04-11T19:05:25.996+05:30: Access control is not enabled for the database. Read and write access to data and
 configuration is unrestricted
        2021-04-11T19:05:25.997+05:30: This server is bound to localhost. Remote systems will be unable to connect to th
is server. Start the server with --bind_ip <address> to specify which IP addresses it should serve responses from, or wi
th --bind_ip_all to bind to all interfaces. If this behavior is desired, start the server with --bind_ip 127.0.0.1 to di
sable this warning
        2021-04-11T19:05:26.539+05:30: Document(s) exist in 'system.replset', but started without --replSet. Database co
ntents may appear inconsistent with the writes that were visible when this node was running as part of a replica set. Re
start with --replSet unless you are doing maintenance and no other clients are connected. The TTL collection monitor wil
l not start because of this. For more info see http://dochub.mongodb.org/core/ttlcollections
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

Step 2: Creating Database

You can first create a test database ‘testdb’ by leveraging the ‘use’ command.

```
> use testdb;
switched to db testdb
>
```

Next, if the database exists then the above command will use that database, or else it will generate a new database.

You can now create a collection called ‘products’ inside this database by using the command ‘createcollection’.

```
> db.createCollection("products")
{ "ok" : 1 }
>
```

Now, you can insert the test documents within the collection with the help of the command ‘InsertMany’.

```
> db.products.insertMany( [
...       { product: "sc1", price:500,colour : "black",brand : "Samsung",size: { height: 16.26, unit: "cm" },available : "yes"},
...       { product: "sc2", price:400 ,colour : "red",  brand : "Samsung",size: { height: 13.3,unit: "cm" },available : "yes"},
...       { product: "sc3" , price:200,colour : "black",brand : "Samsung",size: { height: 11.1, unit: "cm" },available : "no"},
...       { product: "ac1", price:800 ,colour : "black",brand : "Apple",size: { height: 16.26, unit: "cm" },available : "yes"},
...       { product: "ac2", price:600 ,colour : "red",  brand : "Apple",size: { height: 13.3,unit: "cm" },available : "yes"},
...       { product: "ac3" , price:300,colour : "black",brand : "Apple",size: { height: 11.1, unit: "cm" },available : "no"}
...    ] );
{
       "acknowledged" : true,
       "insertedIds" : [
              ObjectId("6072fda8e13afa2f49c7afa9"),
              ObjectId("6072fda8e13afa2f49c7afaa"),
              ObjectId("6072fda8e13afa2f49c7afab"),
              ObjectId("6072fda8e13afa2f49c7afac"),
              ObjectId("6072fda8e13afa2f49c7afad"),
              ObjectId("6072fda8e13afa2f49c7afae")
       ]
}
>
```

Step 3: Creation of Aggregation Pipeline

In the above collection, Let’s say we need to find out the total amount of sales that happened for each of the brands Apple and Samsung for the availab phone. So first we need to filter out the documents based on the available = “True”. This is done using the command **“Match”** and also the first stage

the Aggregation Pipeline. Then we need to find the sum of “price” which is the second stage as shown below**.** In the second stage, the grouping done based on the brand and then the total sum of the price is calculated using the command **“Group”**

```
> db.products.aggregate([
...     { $match: { available: "yes" } },
...     { $group: { _id: "$brand", total: { $sum: "$price" } } }
... ])
{ "_id" : "Apple", "total" : 1400 }
{ "_id" : "Samsung", "total" : 900 }
>
```

You can add one more stage to this output called sort to display the sum based on higher price to lower price as shown below. You can use “sort” for th situation. In the sort, 1 refers to ascending order and -1 refers to descending order.

```
> db.products.aggregate([
...     { $match: { available: "yes" } },
...     { $group: { _id: "$brand", total: { $sum: "$price" } } },
...     { $sort : { total : 1} }
...
... ])
{ "_id" : "Samsung", "total" : 900 }
{ "_id" : "Apple", "total" : 1400 }
>
```

Examples of MongoDB Aggregation Pipelines:

If you consider this test “posts” collection:

```
{
   "title" : "my first blog",
   "author" : "John",
   "likes" : 4,
   "tags" : ["angular", "react", "python"]
},
{
   "title" : "my second blog",
   "author" : "John",
   "likes" : 7,
   "tags" : ["javascript", "ruby", "vue"]
},
{
   "title" : "hello city",
   "author" : "Ruth",
   "likes" : 3,
   "tags" : ["vue", "react"]
}
```

$group

This is what $group would look like on this:

```
db.posts.aggregate([
    { $group: { _id:"$author", titles: { $push:"$title"}} }
])
```

The output for this command would be as follows:

```
{
    "_id" : "Ruth",
    "titles" : [
        "hello city"
    ]
},
{
```

```
    "_id" : "John",
    "titles" : [
        "my first blog",
        "my second blog"
    ]
}
```

$match

This is what the command would look like for $match:

```
db.posts.aggregate([
    { $match: { author:"John"} }
])
```

This is what the result would look like for this command:

```
{
    "_id" : ObjectId("5c58e5bf186d4fe7f31c652e"),
    "title" : "my first blog",
    "author" : "John",
    "likes" : 4.0,
    "tags" : [
        "angular",
        "react",
        "python"
    ]
},
{
    "_id" : ObjectId("5c58e5bf186d4fe7f31c652f"),
    "title" : "my second blog",
    "author" : "John",
    "likes" : 7.0,
    "tags" : [
        "javascript",
        "ruby",
        "vue"
    ]
}
```

$sum

For this example set, we can execute this command as follows:

```
db.posts.aggregate([
   { $group: { _id: "$author", total_likes: { $sum: "$likes" } } }
])
```

This is what the output of this command would look like:

```
{
   "_id" : "Ruth",
   "total_likes" : 3
},
{
   "_id" : "John",
   "total_likes" : 11
}
```

How to Boost MongoDB Aggregation Pipeline Performance?

Here are a few simple things to consider to boost your MongoDB Aggregation Pipeline performance:

- The db.aggregate() command can either store the results in a collection or return a cursor. When returning a cursor or storing the results within a collection, each document in the result set is subject to the BSON Document Size Limit (16 MB currently). Therefore, if any single BSON document exceeds the BSON Document Size Limit, the command will throw an error.
- If you have multiple pipeline stages, it is usually better to understand the overhead attached to every stage. For example, if you have both the $match and $sort stage in your pipeline, it is highly recommended that you utilize a $match before $sort to minimize the documents that you wish to sort.

What are the Limitations of MongoDB Aggregation Pipelines?

Despite the various advantages of leveraging MongoDB Aggregation Pipelines for your business use case, it is far from perfect. As far as limitations a concerned, the result has the same size limitations per document (16 megabytes). On top of this, every stage is limited by 100 MB of RAM.

You can work around the size limitations by leveraging the allowDiskUse option, otherwise, MongoDB might throw an error.

Conclusion

In this module we studied

- Limit and skip in mongodb for pagination
- Indexes in mongodb
- Aggregation Pipeline in mongodb

Interview Questions

> **Explain the term “Indexing” in MongoDB.**

```
In MongoDB, Indexes help in efficiently resolving queries. What an Index does is that it stores a small part of the data

Indexes look at an ordered list with references to the content. These in turn allow MongoDB to query orders of magnitude

For example:

`> db.users.find({"username": "user101"}).explain("executionStats")`

Here, `executionStats` mode helps us understand the effect of using an index to satisfy queries.
```

> **What is the Aggregation Framework in MongoDB?** - The aggregation framework is a set of analytics tools within MongoDB that allow you to do analytics on documents in one or more collections. - The aggregation framework is based on the concept of a pipeline. With an aggregation pipeline, we take input from a MongoDB collection and pass the documents from that collection through one or more stages, each of which performs a different operation on its inputs (See figure below). Each stage takes as input whatever the stage before it produced as output. The inputs and outputs for all stages are documents—a stream of documents.

Thank You !