# INDEX

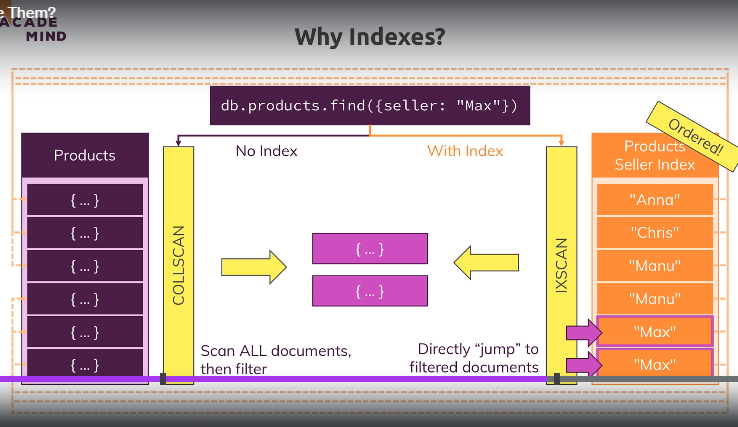
An index is an ordered list of all values of an indexed field

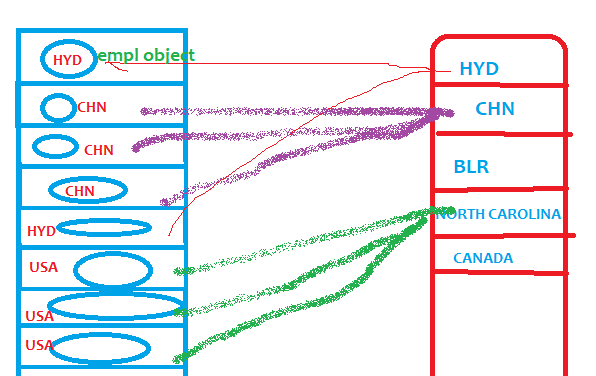
This is also same like book, in the index page u will have the page number of all the chapters, so that if any requirement comes first u have to see the index page and move on to the page which ever u want.

Creating Index is to reduce the searching time/ for fast retrieval of data, index search is faster in mongo indexes are created in something like separate collection,

Same like book, every element in index have the reference /page number , same way every element in index points to the document in the original element in the collection

Indexes won’t come for free





Why index is fast and what happens after creating indexes???

After creating index let’s say on employee address, all the values (address, lets say) will be stored in a separate collection, and each value will be pointing to the original document so after creating index, scan will be performed on index values called index scan ,

Scan will not be performed on the entire collection (called collection scan)

Here one more advantage is the values of the index are sorted , so searching in these indexes is also very easy

Use case 1:-

Consider we are storing payments made by each person-1 customer can do 100’s of payments

Each person will have 1 account number, if we create index on account number, it will be best idea,

Lets a customer made 10,000 payments, then as usual all values (all account numbers)will be stored in index table and that account number will be linked to his multiple payment documents

When we search with that account number, first it will go to index table straight away to that account

New Order(20$,”AcNum582”)

New ORder(20$,”AcNum123”)

New ORder(20$,”AcNum582”)

number it will fetch all those his payments

New ORder(20$,”AcNum123”)

The above u are seeing is an index in right side, index is created on account numbers

1. Then all the account numbers alone will be stored in some ordered collection only account numbers not entire collection , as mentioned in right side
2. Next time when we are searching with account number first it wont go to collection instead it will goto separate collections where index values are stored ,here all the index values are stored in separate collection so control will come here as there are already ordered

### **Why indexing is so faster**

1. All indexed values are stored in separate collection
2. Each value will have references to the original document , so that first index scan will be performed and say if we found **acnum99** now this value have all the references to the original doucments,if on this account 10,000 payments are done all 10,000 documents references will have mapped to this value index, so it will straight away go with reference search and hence searching is very easy
3. After creating indexes, It will use index scan instead of collection scan, once desired index is found it will return all the pointers to the main documents in the original collection

Index creation is costlier

It’s costlier because separate index table will be created, and each index value will be mapped to a document present in the original collection

If no index, then record creation will be easy as it straight away inserts into same collection

If u create index, then insertion should happen in 2 tables-1 main table & index table

Since index creation will results in faster searching ,don’t create indexes on each and every column

EX:- lets say we have employee collection (Name:’Mani’,age:20.Address:”hyd”) here if u create index on every column , then three tables will be created like name,age,address everytime a new record comes the entries must be present in these 3 columns also,

1. name field will be pointing to original document,
2. age field also will have the reference to the original document
3. address also have the reference to original document

HERE the main problem is if u create the index on all columns , of course fast retrival but insertion will be slower

## Queries for revealing performance/exec plan

Db.persons.explain(“executionStats”).find(“dob.age”:{$gte:60});

db.persons.explain('executionStats').find({'dob.age':{$gte:60}});

db.contacts.explain().find({'dob.age':{$gte:60}});

db.contacts.explain().find({'dob.age':{$gte:20}}).limit(1);

RESULT

executionStats: {

executionSuccess: true,

nReturned: 1328,

executionTimeMillis: 16,

totalKeysExamined: 0,// HERE it is saying keys examined is zero, means no index

totalDocsExamined: 5000,//it scanned only documents

executionStages: {

stage: 'COLLSCAN',// it performed entire collection scan index instead of index scan

filter: { 'dob.age': { '$gte': 60 } },

nReturned: 1328,

executionTimeMillisEstimate: 1,}

It will show summary+winningPlan+plan decision process

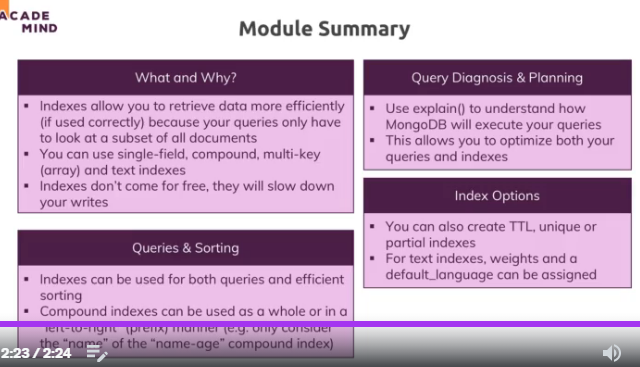
It will show summary+winning+

RejectedPLan

It will show summary+winningPlan

There are 2 types of scans

1. Collection Scan (COLLScan)- this is worst performing entire collection scan
2. Index scan (IXScan) –this is Best



## Creating indexes

db.contacts.createIndex({'dob.age':1});

by default this is a foreground index-faster, where as collection is locked during index creation

same like when ur writing index page for the book , u will stop writing data into the book, this is also same way

u can create background index also as

db.contacts.createIndex({‘dob.age’:1},{background:true}); //BS- Back ground indexes are slower, but it wont lock the collection while indexing

after creating indexes using below query

db.contacts.explain().find({'dob.age':{$gte:20}}).limit(1);

### Creating Unique Indexes

db.contacts.createIndex({email:1},{unique:true});

same like oracle if u want to apply unique constraint on a column, create an index with that flag simple

### Creating compound indexes

31-Male

32-Male

33-Female

34-Female

34-Male

Means only 1 index will be created on 2 columns, only 1 index

Lets say if we create index only age and gender

db.contacts.createIndex({'dob.age':1,gender:1});

dob.age\_1\_gender\_1

#### Rules of Compound indexes

Note:- You created compound index with age+gender

1. If u search with 1st parameter age 🡪 then Index scan (**IXscan**) will be performed, because if you see index values are having starting letters as age and index values are sorted with age

Db.contacts.find({age:{$gte:40}})🡪 **then Index scan** will be performed

1. If u search with 2nd param-gender 🡪entire collection scan (**COLLSCAN**) will be performed, because index values are not sorted with gender, those are sorted with age.
2. If u search with both age+gender then, index scan will be performed. Because it will try to search with age, as with age which is first parameter only index values are sorted

**db.passengers.explain('executionStats').find({age:35,name:'Alexis Bohed'}):// IXscan**

Note: While creating compound index consider creating partial index (means index created on partial data)

And if u don’t want creating index on 2 fields –like age,gender, create partial index on data which u use frequently

Sample index values will look like above,

Case study

1. Searching will be faster when We can combine ly search with 2 fields(index scan will happen)
2. It will be faster when searching happened with (1st parameter) age alone(index scan will happen)
3. It will be slower when we search with gender because Collection scan will happen hence slower ,it will prefer collection scan because we are searching with 2nd field , it prefer collection scan because values are sorted with first fieldif we search with 1st field index scan will happen

shop> db.contacts.explain('executionStats').find({'gender':'male'});

{ //here we are searching with second parameter of the compound index-age+gender

explainVersion: '1',

queryPlanner: {

namespace: 'shop.contacts',

indexFilterSet: false,

parsedQuery: { gender: { '$eq': 'male' } },

queryHash: '025F03D3',

planCacheKey: '4B09AA47',

maxIndexedOrSolutionsReached: false,

maxIndexedAndSolutionsReached: false,

maxScansToExplodeReached: false,

winningPlan: {

stage: 'COLLSCAN', //here see the entire collection scan happened instead of index scan

filter: { gender: { '$eq': 'male' } },

direction: 'forward'

},

rejectedPlans: []

},

### Partial indexes

It is creating indexes on partial data, instead of creating indexes on entire collection data, only we will create index on certain amount of data

Let’s we have

And if u don’t want creating index on 2 fields –like age,gender, create partial index on data which u use frequently

db.contacts.createIndex({'dob.age':1},{partialFilterExpression:{'dob.age':{$gte:50}}});

db.email.createIndex({email:1},{unique:true,partialFilterExpression:{email:{$exists:true}}})

Now here index will be created only on documents whose age is >50,

shop> db.contacts.explain().find({'dob.age':{$gte:40}});

{

queryPlanner: {

namespace: 'shop.contacts',

parsedQuery: { 'dob.age': { '$gte': 40 } },

winningPlan: {

stage: 'COLLSCAN', //we created a partial index –means index created on partial data , here it is collection scan because we created index on data only greater than 50 & but we searched for age>40 on these data index is not created hence index scan was not possible and created collection scan

filter: { 'dob.age': { '$gte': 40 } },

direction: 'forward'

},

db.contacts.explain().find({'dob.age':{$gte:51}});

{

explainVersion: '1',

queryPlanner: {

parsedQuery: { 'dob.age': { '$gte': 51 } },

winningPlan: {

stage: 'FETCH',

inputStage: {

stage: 'IXSCAN',// its partial index, here it is index scan because we created index on partial data whose age is greater than 50 .as we are searching for age>50 it performed index scan

keyPattern: { 'dob.age': 1 },

indexName: 'dob.age\_1',

Usecases

1. Lets say u want to create index on email field ,but u have many documents which have null values ,since having many null values those are considered as duplicates and hence unique index creation fails
2. Now we can create unique index on partial data, u can exclude those Bloody null documents u can create unique index on remaining documents

shop> db.email.find().pretty();

[

{ \_id: ObjectId("64297be8d825310fdd6a1835"), name: 'Manideep' },

{ \_id: ObjectId("64297be8d825310fdd6a1837"), name: 'charan' },

{\_id: ObjectId("64297be8d825310fdd6a1836"),name: 'santu',email: 'santu@gmail.com'},]

If u see above, there was 2 documents email having null value(which is not unique)

if we try to create unique index it will fail , saying documents are not unique

db.email.createIndex({email:1},{unique:true});

**MongoServerError: Index build failed: b343efe4-a28e-4b59-bdac-051014d5f9ba: Collection shop.email ( cf73b97b-22f2-4a99-b6f8-171e1432f12e ) :: caused by :: E11000 duplicate key error collection: shop.email index: email\_1 dup key: { email: null }**

**In this case, we have to filter out null data and apply unique index on remaining records**

db.email.createIndex({email:1},{unique:true,partialFilterExpression:{email:{$exists:true}}})

### **TTL (Time to live) Indexes**

When u create TTL index, records will be deleted automatically after some specified time

And index must be created only on date fields,why??? Because , in a collection we can have many date fields like createdDate,lastModifiedDate,ExpiryDate all these columns are date type only, but mongo doesn’t know based on which column it should delete data

Hence creating index on date column is mandatory, even if u create on text column it wont work

And it wont work on compound indexes

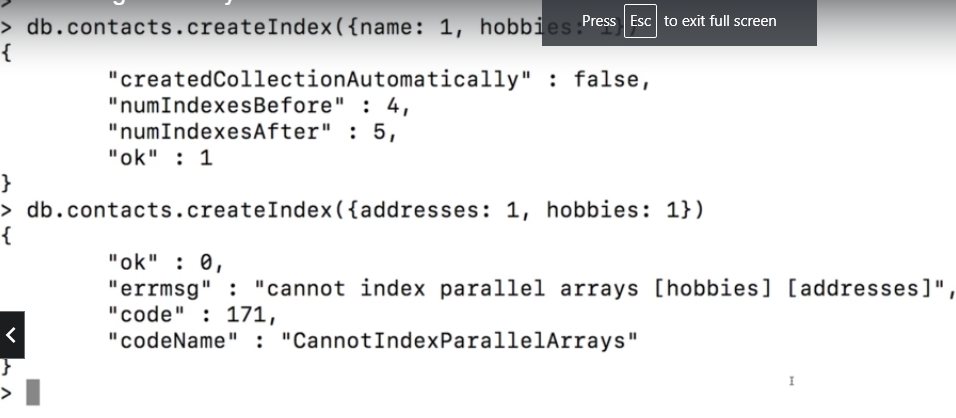
db.email.createIndex({createdDate:1},{expireAfterSeconds:10});

db.eventlog.createIndex( { **"lastModifiedDate"**: 1 }, { expireAfterSeconds: 3600 } )

### Multikey indexes

This is creating index on array column like hobbies

If each array contains 4 unique value, if we have 1000 documents, then 4,000 unique values will be stored in index table, each value will be pointing to the original documents



### Text Indexes

We can create index on text field –“db.books.createIndex({description:'text'});”

1. **db.ships.createIndex({chart:'text'})**

only difference in syntax is ,generally we will give column name:1 or -1, which indicates ascending or descending order,here so I have a doubt then will keys stored in ascendin or desc order

How this text index works



When u create text index on the text column, then the contents of that sentence will be split into small words will be removed and main words will be kept and inserted in separate index table , which inturn points to original documents

You can have only create 1 text index for 1 collection

1. **db.books.find({$text:{$search:'colour'}});**

Here u don’t need to mention the column name, because text index creation only possible on 1 column in one collection

this will fetch only documents who have the content called “colour”

[ { \_id: ObjectId("643a7e9d6b3203e2a32813ee"), name: 'spring',

description: 'This book is almost equally important book its in red colour '

}]

2)

1. shop> db.books.find({$text:{$search:'yellow colour'}});

[ {

name: 'spring',

description: 'This book is almost equally important book its in red colour '

},

{

name: 'java8',

description: 'This book is really fantastic and its most truested right source and its Yellow '

}]

//Here the problem is I have searched for **yellow colour where as it searched for documents having keyword as yellow and keyword called colour**

T

To avoid this problem use below query to fetch the document having yellow colour

**db.books.find({$text:{$search:"\"yellow colour\""}});**

1. **db.ships.countDocuments({$text:{$search:'DNC'}});**

remember the query like text search

now this will fetch only for yellow colour ,but it cant find hence it doesn’t any records

Note:- always while searching if u use any wild card character then it must be followed by double quotes , above we used \ back slash and it must be followed by “

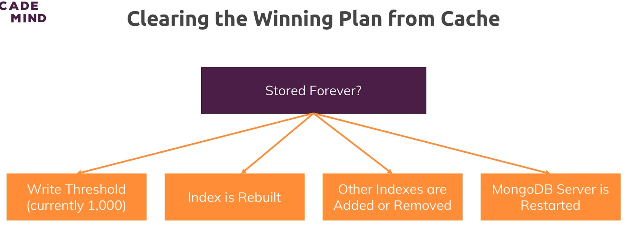
#### **Creating compound text indexes on multiple columns**

The main problem is u can create only 1 text index per collection,

Hence u can’t create second text index in the same collection, then we can create compound text index on 2 columns

**db.books.createIndex({description:'text',name:'text'});**

## Choosing winning plan



Generally it will caches the winning plan for same type of queries, so that if same query comes again it will calculate/decide the winning plan again, it will just get the plan from cache and uses that

1. After storing 100 docs it will again clears the cache and calculate the winning plan again
2. When other existing indexes are removed then also winning plan from the cache will be removed
3. When mongodb server is restarted also then cache will be cleared

## Dropping indexes

db.contacts.dropIndex({'dob.age':1}); db.contacts.dropIndex({'dob.age':1,gender:1})

db.ships.dropIndex('chart\_text');//dropping a text index is bit difficult , we have to drop by name

to get the index name type “db.ships.getIndexes();”

## Show the indexes

db.contacts.getIndexes();

## Indexes Behind the Scenes

What does createIndex() do in detail?

Whilst we can't really see the index, you can think of the index as a simple list of values + pointers to the original document.

Something like this (for the "age" field):

(29, "address in memory/ collection a1")

(30, "address in memory/ collection a2")

(33, "address in memory/ collection a3")

The documents in the collection would be at the "addresses" a1, a2 and a3. The order does not have to match the order in the index (and most likely, it indeed won't).

The important thing is that the index items are **ordered** (ascending or descending - depending on how you created the index). createIndex({age: 1}) creates an index with **ascending sorting**, createIndex({age: -1}) creates one with **descending** **sorting**.

MongoDB is now able to quickly find a fitting document when you filter for its age as it has a sorted list. Sorted lists are way quicker to search because you can skip entire ranges (and don't have to look at every single document).

Additionally, sorting (via sort(...)) will also be sped up because you already have a sorted list. Of course this is only true when sorting for the age.

## Corner scenarios

Creating indexes on Boolean values may not help

Because in index table only 2 values will be there , so we cant filter more documents- because with this only half of the documents are filtered out

Creating index on text field will also work –ex:- address-

### Use indexes for sorting

INDEX:- separate collection of only values of the column which index is created, every value will internally point to those documents present in the original collection

If u have a huge collection then sorting will be difficult

Whereas if u have an index already on it , if u have already index on account number , then all those values would have been already been sorted and stored in separate collection (c), then if u want to sort those values while fetching if index is already there then it will be easy to fetch, because 1st it will go to indexed values in separate collection as those indexed values are already sorted and each value pointed to original document it will simply go key by key

Hence if u are sorting based on the column, better to have index on the column on which u wanted to do sorting