MongodB

- No sql, non relational database
- eg for no sql databases : mongodB, Casandra, couchdB
- Collection of data, document.

→ By default available dB's are:

admin
config
local

→ To create our own dB

use "database name"

→ simply type db - ഇതാണ് database ഉണ്ടോന്ന്
ഉറപ്പിക്കാം.

→ In mongodB datas stored in document type, and datas stored as key value pairs

advantages:

→ documents in mongodb is very much like json or JS objects.

→ It allows us to store nested documents within a document

eg: a doc in the books collection could have an authors property and the authors property could be a doc in itself which has a first

name property, last name property and age property. This will be a alternative method compared to sql where it has 2 tables for different properties.

→ Allow our data to be little more flexible

→ High speed

MongodB shell - allows us to interact with mongodb from a terminal

gui - mongodb combass to visualize whats going on

collections & documents

```
{"title": "my first blog",
"author": "Rubeena",
"tags": "["video games", "review"],
"upvotes": 20,
"body": "lorem ....."
}
```

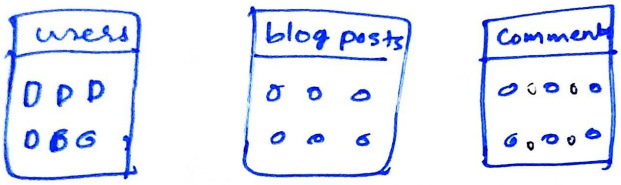
array of strin

→ Here every document have a unique id property

to identify them. This property would be a special object id type in mongodb and its assigned to the document by mongodb itself when we create a document. ~~When~~ we can use that document by that unique Id

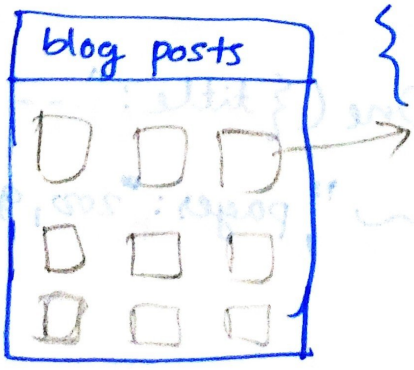
- In mongodb there is different collections ~~of~~ are there for different types of data.

eg:



- Inside each of the collections, there is we'd store a load of

- In mongodb documents looks very much like a jsons object with key value pairs but actually its ~~been~~ stored as something called bson which is just binary json ~~but for all intents and purpose a document~~




```
{ "title" : "my blog",        ← strings
  "author" : "Rubeena",
  "tags" : ["games" "reviews"],   ← array of strings
  "upvotes": 20,
  "body" : "Lorem..... " }
```

to mongodb compass

→ local database contains startup log data
→ This is showing us all of our databases that we create, all the of the collections in those db and all of the documents in those as well.
→ C R U D documents is possible here.

mongodb shell

→ show dbs
→ use
→ use test
→ use mydb ('mydb' name new db create or use)
→ db (which db you're currently running in)
→ show collections
→ help

Adding documents:

if collection is bookstore and doc is books,

→ use bookstore
→ db.books.insertOne({title: "-----",
author: "~~~", pages: 200, genres: ["~~",
})

bookstore> db.authors.insertOne({name:"——",
age:60})

→ db.books.insertMany([{title:——},{——}])

→ bookstore> db.books.find()

To display first 20 books from the bookstore.

→ db.books.find({author:"——"})

To display that authors books only.

→ db.books.find({author:"——",rating:7})

To display authors is " " and the rating is 7.

→ db.books.findOne({_id:ObjectId("——")})

To find the id based search

Sorting & limiting data

→ Count the amount of docs we would get back from this query

db.books.find().count()

→ db.books.find({author:"_____"}).count()

count the no. of books of that author

→ db.books.find().sort({title:1})

1 - means sorting in ascending order

-1 - sorting in descending order

→ we can also if we add '.limit(3)' to that so sorting and display only 3.

Nested documents

→ db.books.insertOne({title:"_____", author:"_____", rating:9, pages:500, genres:["fantasy"], reviews:[{name:"_____", body:"Great book"}, {name:"mario", body:"_____"}]})

→ If you want to insert many items:

db.books.insertMany([{title:"_____", author:"_____", reviews:[{name:"_____", body:"_____"}, {name:"_____", body:"_____"}]}, {title:"_____", reviews:[{name:"_____",

body: " " }]}, etc . . . . . }]}]).

This is how we can add nested documents, to our documents inside a collection

complex queries and operators

→ db.books.find({rating:7})

by using this method, it would find all the books which have a rating of exactly 7.

→ Sometimes we want to find any rating less than 4, so for this we can use a special query operator. Queries in mongodb is used with a '$' sign.

→ To find all the books that have a rating greater than 7, we can use

db.books.find({rating:{$gt:7}})

→ To find books, which are less than 8,

bookstore> db.books.find({$lt:8})

→ for less than or equal to use ({$lte:8})

gte - greater than equal to

→ db.books.find({rating:{$gt:7}, author:"..."})

for using 2 or more filters together

→ OR:

db.books.Find({$or:[{rating:7},{rating:

⇒ db.books.find({$or:[{rating:7},{author:

for getting a rating of 7 or author is ___

⇒ db.books.find({$or:[{pages:{$lt:300}},

{pages:{$gt:400}}]})

→ Querying arrays:

⇒ in:

This 'in' operator to say okay ~~well~~ for a particular field is going to be within a certain range of values. If you like or a certain array of values ~~now that might go over your head at first but once~~

⇒ If we wanted to filter a rating 7 or 8 or 9. we can use 'or' but instead of that we can use 'in'.

db.books.find({rating:{$in:[7,8,9]}})

⇒ $nin (not in)

db.books.find({rating: {$nin: [7,8,9]}})

means to display the value which is not 7, 8, or 9.

⇒ To get a special feature from the array of genres

db.books.find({genres: "fantasy"})

⇒ db.books.find({genres: ["magic"]})

to find the array which only has 'magic' as genres.

⇒ db.books.find({genres: ["fantasy", "magic"]})

to display the books collection which has fantasy and magic as genres.

⇒ To check an array field to see if all of a specified list of items is in the array

db.books.find({genres: {$all: ["fantasy", "magic"]}})

⇒ To check any book where there is a review for that book by 'Ruskinbond'

db.books.find({"reviews.name": "Ruskinbond"})

Deleting documents

⇒ 'deleteOne' can used to delete single document
⇒ 'deleteMany' to delete many documents.

⇒ ~~db~~ deleteOne.

db.books.~~find~~(deleteOne({_id: ObjectId("...")})

⇒ deleteMany.

db.books.deleteMany({author:"____"})

Here delete all documents which has the Author "____"

⇒ • To find all the documents,

db.books.find()

Updating documents

⇒ for updating a document,

db.books.updateOne({_id:ObjectId("...")}, {$set:{rating:8, pages:360}})

update filter

first argument, id of the doc

2nd argument

changes should type here.

⇒ ~~db.books.find(updateOne~~ Many.

⇒ db.books.updateMany({author: "①"}, {$set: {author: "②"}})

Here the author: 1 is changed to author name 2.

⇒ for incrementing any of the fields we can use inc

db.books.updateOne({_id: ObjectId("------")}, {$inc: {pages: 2}})

Here pages will be increased by 2. ie, 300 is to 302 also we can use -2 to decrement pages.

⇒ Pull property - to remove one item from the array

db.books.updateOne({_id: ObjectId("——")}, {$Pull: {genres: "fantasy"}})

⇒ Push - used for insert a new item to the array.

db.books.updateone({_id: OjectId("——")}, {$push: {genres: "fantasy"}})

⇒ each - to add two different genres to the book

```
db.books.updateOne({_id: ObjectId("_____")},
    {$Push: {genres: {$each: ["1","2"]}}})
```

Here pushing each of the array to the document. So the final op is:

genres: ["fantasy", "comedy", "1", "2"]

Mongodb drivers

- Drivers are used to communicate with the mongodb programming langs such as node.js, python, ruby etc.
- eg for drivers are
- Node drivers are used to communicate b/w node applications and mongodb

connecting to mongodb

- created db.js for database fnctns
- In this file, there will be two fnctns
  - → initially connect to a db
  - → to retrieve that db connectn once we

already have connected to it

- module.exports – to export properties in node.
- connectToDb: () – the job of this fnctn is going to be to initially connect to a db
- getdb: () – this is the 2nd fn we already told

```
const {mongoClient} = require('mongodb')
module.exports = {
  connectionToDb: (cb) => {
  MongoClient.connect('mongodb://localhost:
                       27017/bookstore')
    .then((client) => {
      dbConnection = client.db()
      return cb()
    })
    .catch(err => {
      console.log(err)
      return cb(err)
    })
  },
getDb: () => dbconnection
```

promise

db name

mongodb with node.js

```
const mongoclient = require ("mongodb").MongoClient
//to point mongodb to our database
const connection_String = "mongodb://localhost:27017/"
```

27017 is the port where mongodb runs

```
MongoClient.connect (connection_String, {useUnifiedTopology: true}, function
(err, client){
    if (err) throw error;
    console.log ("connected to mongodb")
    Var db = client.db ("bookstore")
```

connection_String → connection URL
function → CB
(err, client) → either error or proper connection
db → to connect to a db which we created

```
db.collection ("books").findOne ({}, function (err, result){
    console.log (result)
    client.close()
})
```

collection ல் data fetch ஆகிறது
client.close() → data வரும்போது mongodb close ஆகிறது

While doing this, we get only one find element. If we want to find more than one

element we can use find() as,

```
db.collection("books").find({}).toArray(()=>{
    if(err) throw error;
    console.log(result)
    Client.close()
```

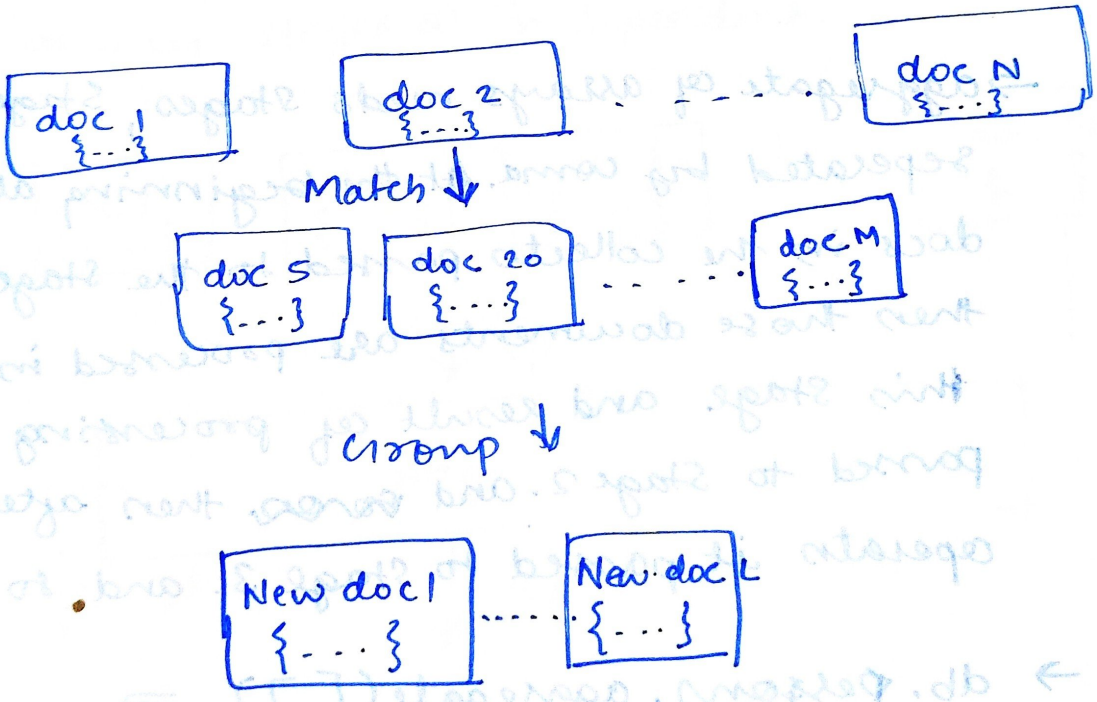
⇒ If we want to display the title only then in node.js we can write,

```
db.collection("books").find({title:'—'},
    Projection:{title:1, _id:0}}).toArray(()=>{
    if(err) throw error;
    So console.log(result)
```

Mongodb aggregation

- This is the most powerful tool that mongoDB offers
- Using aggregation framework you can easily group documents and collections by specific conditions
- Can add additional fields during grouping such as avg, total, min, max etc.
- It is very fast, so that we will get the results fastly

Aggregation process



As a result of grouping finally we will get brand new document

aggregate()

- Aggregation method uses a special method called aggregate
- docs during aggregation process pass through the stages

```
db.<collection>.aggregate([
    <stage1>,
    <stage2>,
    .....
    <stageN>
])
```

→ aggregate of arrays needs stages, stages are seperated by coma. At the beginning all docs in the collects passed to the stage1, then those documents are processed in this stage and result of processing is passed to stage2. and then after each operatn it passed to stage 3. and so on.

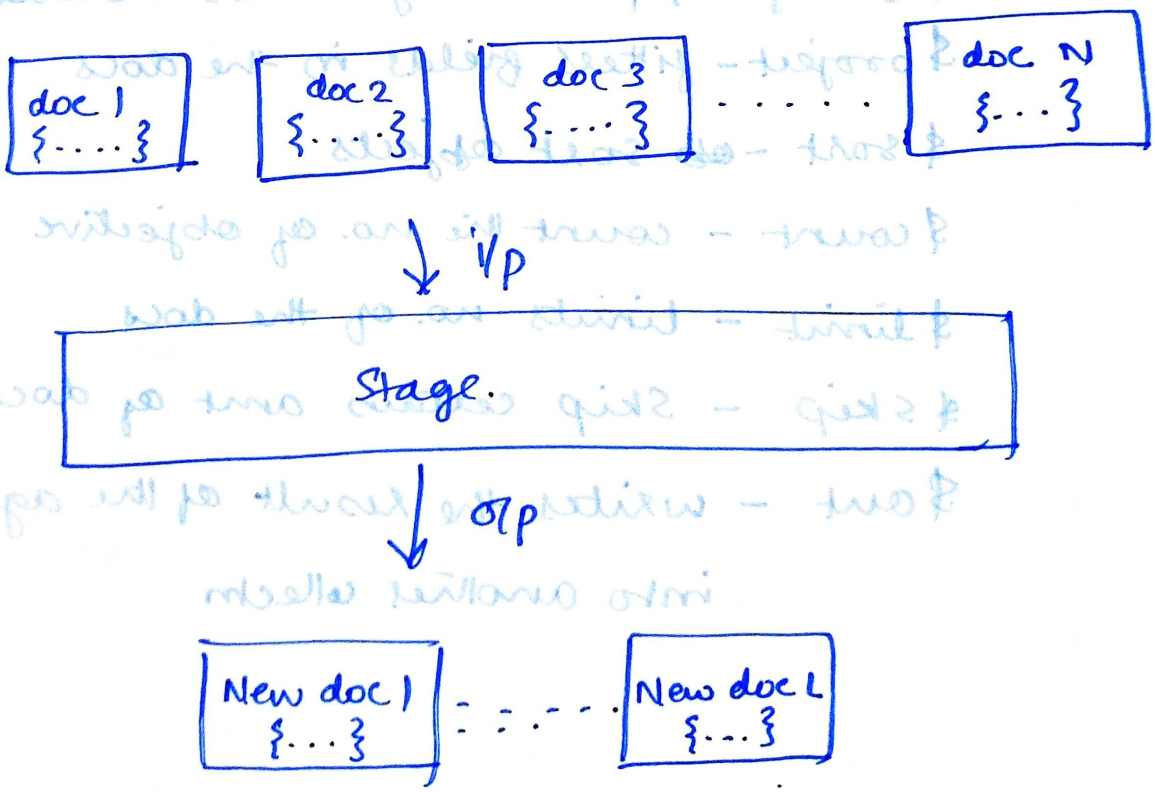
→ db.persons.aggregate([]) =
db.peesons.find({})
both are displaying the same result as out

collections

- Show collections - for showing collections
- Then - use 'collection name' for use the collects

Aggregate Stages overview

→ Agg Stage - is exists in aggrtn framework. Each stage works independent from others. Each stage ~~sta~~ takes doc as i/p then perform its operatn and o/p docs. The resultant doc is reordered or limited to a certain no. or it can be brand new doc when we use. for eg group stage. all I want you to keep in mind is that each stage doesn't impact other stages. One stage is passed to another. Stages are independent.



Aggtn Stage operators

→ Each stage starts from the stage operatr

{ $ < stage Operator > : { } }

eg:

{ $match : { age : { $gt : 20 } } }

{ $group : { id : "$age" } }

{ $sort : { count : -1 } }

→ ~~to~~ Some stage operators used in aggregatn framework

$match - used to filter docs by certain query

$ group - grp docs by certain criteria

$ project - filters fields in the docs

$ sort - ~~ob~~ sort objects

$ count - count the no. of objective docs

$ limit - limits no. of the docs

$ skip - skip certain amt of docs

$ out - writes the result of the aggregatn into another collectn

Aggregation expressions

- Expression refers to the name of the field in i/p docs

  "$<~~fieldname~~ field>"

eg: { $group : { _id : "$age" } }

{ $group : { _id : "$company.location.country" } }

{ $group : { _id : "$name", total : { $sum : "$price" } } }

$match Stage

- match stage is an argument
- match specific docs using query

{ $match : { <query> } }

⇒ db.persons.aggregate ([{ $match : { age : { $lt : 25 } } }])

This will print the ~~pers~~ document, which has a age less than 25.

① db.getcollection ('persons').find ({ tags : { $size : 3 } })

② db.persons.aggregate ([$match : { tags : { $size : 3 } }])

Both progms have same results. As one is run in node.js and other in the terminal

$group stage

- Groups i/p docs by certain expressions

{$group: {_id: <expression>, <field1>:
{ <accumulator1>: <expression1> }, ... }}

- Here keys are field names and values are expressions

eg:

{ $group: { _id: "$age" } }

{ $group: { _id: { age: "$age", gender: "$gender
} } }

- Grp performs group and i/p docs by certain field or fields. Underscore id is mandatory. And it must be present in each grp stage.

⇒ group by nested fields:

db.persons.aggregate([{$group: {_id: $company.
location.country }}])

~~⇒ group by~~

⇒ group by multiple fields

~~$group and $match:~~

⇒ db.persons.aggregate([{$group: {_id: {age:
"$age", gender: "$gender" }}}])

eg: $match & $group:

⇒ db.persons.aggregate([{$match:{favoriteFruit:"banana"}},{$group:{_id:{age:"$age",eyecolor:"$eyecolor"}}}])

result:

{_id:{eyecolor:'green',age:21}},

{_id:{eyecolor:'blue',age:22}},

. . . . .

. . . . .

$group and $match

db.person.aggregate([{$group:{_id:{age:"$age",eyecolor:"$eyecolor"}}},{$match:{favoriteFruit:"banana"}}])

result: Result will be empty. Bcz wrong Stages order. In the 1st stage there is age and eyecolor only. After fetching data the result stage have only age and eyecolor properly. From that we dont get any match for fruit.

⇒ db.peesons.aggregate([{$group:
{_id:{"age":"$age", eyecolor:"$eyecolor"}}},
{$match:{"_id.age":{$gt:30}}}])

result:

{"_id":{"age":38, "eyecolor":"brown"}}

{"age":33,

{"age":32, ...

:

$count stage

- counts no of documents gets as an i/p

{$count:"<title>"}

eg:

{$count:"countries"}

eg:

⇒ db.peesons.aggregate([{$count:"allDocumentsCount"}])

result: {"all documents count: 51}

diff docs count methods

① ⇒ db.persons.aggregate([]).toArray().length

1,7 → 1000 (Client side count)

⇒ Cursor can be used in toArray and forEach methods

⇒ In JS 'Lance' is a property of ~~each~~ ~~Foreach~~ each array and it means that the comment will returns us no of the docs in the cursor

② count method 2.

db.persons.aggregate([]).itcount()

1,4 sec → 1000 (Client side)

itCount - it is a method of the cursor and it will count all docs in the cursor

③ count method ③

db.persons.aggregate([{$count : "total"}])

0,21 sec → {"total" : 1000} (Server side)

Here ① and ② count methods are took long time where as count ③ method is faster than the others. The reason is that ① and ② are client side methods and those cases we iterate through the cursor till the end of the client side. But in the last result server performes count operatn and returns us back just result. It will give us any documents just count thats

why this method is so quick

db.persons.find({}).count()

↓ 0,21 sec ⟶ 1000 (server side)

count helper method

→ Find count() is wrapper of the aggregate $count

⇒ eg: $group and $count

db.persons.aggregate([{$group:
{_id:"$company.location.country"}},
{$count:"countriesCount"}])

ans: {"countriesCount":4}

⇒ db.Persons.aggregate([{$group:{_id:
"$eyecolor"}},{$count:"eyecolor"}])

Here there are 3 different eyecolors are there.

result = 3

⇒ db.persons.aggregate([{$group:
{_id:{eyecolor:"eyecolor", gender:
"$gender"}}},{$count:"eyecolor and gender"}
])

result: { "eyecolor and genders": 6 }

here genders = 2, eyecolor = 3

∴ total = 6 combinations

$Sort Stage

- Sorts i/p docs by certain field(s)
- Usually the sort stage is placed after the match and group stages

{$Sort: { <field>: <-1/1>, <field 2>: <-1/1>; .... }}

-1 - descending

1 - ascending

eg: { $sort: {$score: -1} }

{ $sort: { age: 1, country: 1 } }

- Sort stage performs reordering of the i/p docs. Sort order is specified using key value pairs, where value is either 1 or -1

db.Persons.aggregate([{$sort: {name: 1}}])

db.persons.aggregate([{$sort: {name: 1, age: 1, gender: -1, eyecolor: 1}}])

⇒ Group & sort

- db.persons.aggregate([{$group:
  {_id:"$favoriteFruit"}}, {$sort:{_id:1}}])

- db.persons.aggregate([{$match:
  {eyecolor:{$ne:"blue"}}}, {$group:
  {_id:{eyecolor:"$eyecolor"}},
  {$sort:{"_id.eyecolor":1,"_id.favoritefruit":-1}}])

$project

- Includes, excludes or add fields

{$project:{<field1>:<1>, <field2>:<0>,
<newField1>:<expression>}}

1 - field included

0 - field excluded

expression - rename field to the new field one

eg:

{$Project:{name:1,"company.title":1}}

{ $project : { _id : 0, name : 1, age : 1 } }

{ $project : { eyecolor : 0, age : 0 } }

{ $project : { name : 1, newAge : "$age" } }

eg :

db.persons.aggregate([{ $project : { name : 1,
"company.location.country" : 1 } }

This means it will take i/p docs, leave only 3 variables _id, name, company.lo.cou. In $project '_id' field will be a result even if we are not calling it. If we give "_id : 0", then there wont be any id in the result field

$project with new ~~eg~~ fields.

db.person.aggregate([{ $project : { _id : 0, name : 1, info { eyes : "$eyecolor", fruits : "$favoritefruit", country : "$company.location.country" } } }])

$Limit stage

- o/ps first N docs from the i/p

  { $limit : <number> }

  eg: { $limit : 1000 }

It usually occurs in 2 cases

1. Sampled aggregation reqs with $limit as 1st stage
2. After $sort to produce topN results

$limit, $match & group

```
db.persons.aggregate([{$limit:100},
  {$match:{age:{$gt:27}}},
  {$group:{_id:"company.location.country"}
                    {$limit:100},  ])
db.persons.aggregate([{$match:{eyecolor:
{$ne:"blue"}}},{$group:{_id:{eyecolor:
"$eyecolor", favoriteFruit:"$favoriteFruit"}}}
{$sort:{"_id.eyecolor":1,"_id.favoritefruit":-1}}
```

$unwind Stage

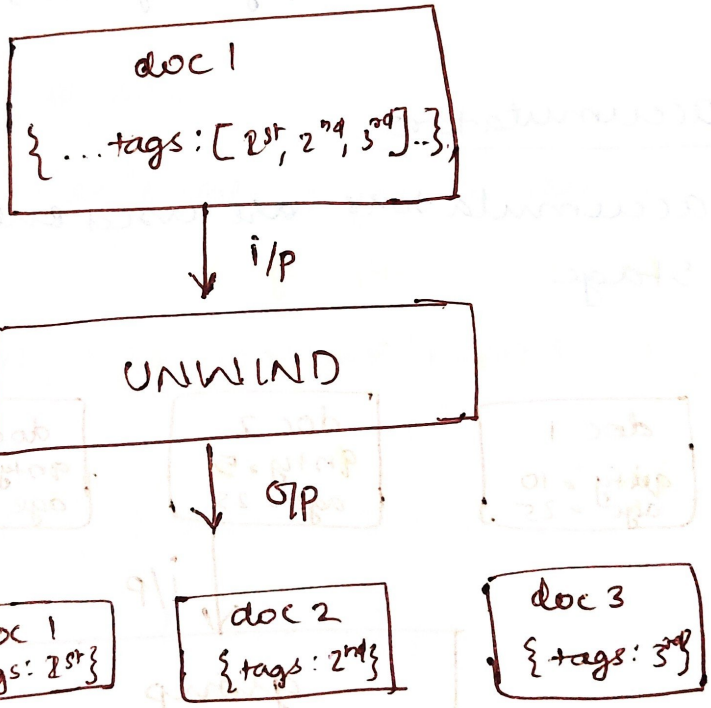
- Splits each doc with specified array to several docs - one doc per array element

{ $unwind : <arrayReferenceExpression> }

eg: { $unwind : "$tags" }

{ $unwind : "$hobbies" }

unwind used in arrays



- Here the array is changed to text string.
- If the i/p doc contains 15 docs, we will get unwind result as 15 docs

eg: $unwind & $project

```
db.persons.aggregate([{$unwind:"$tags"},
{$project:{name:1, index:1, tags:1}}])
```

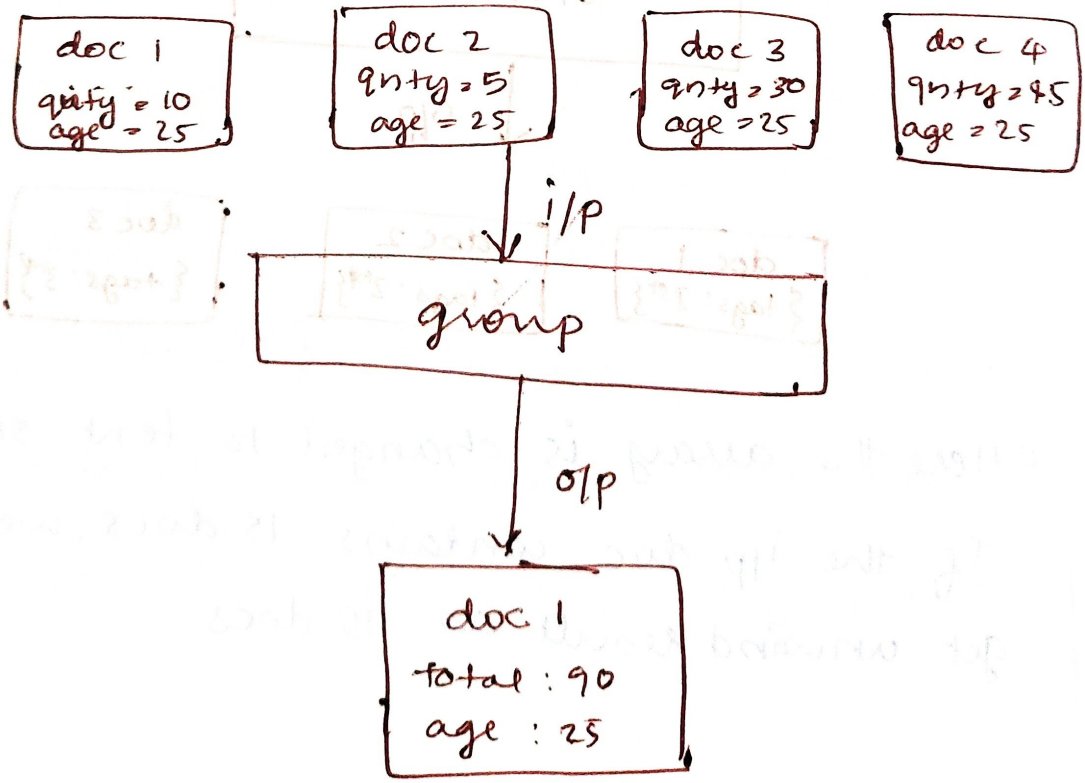
⇒ unwind & $ group

```
db.persons.aggregate([{$unwind:"$tags"},
{$group:{_id:"$tags"}}])
```

result : only tags with diff docs

$accumulators

- accumulators are used only in the $group stage



- Accumulators maintain state for each grp of the docs

  { $accumulatorsOperator : <expression> }

eg:

{ $sum : "$quantity" }

{ $avg : "$age" }

{ $max : "$spentmoney" }

{ $min : "$spentmoney" }

Sum accumulators

- Sums numeric values for the docs in each group

  { $sum : <expression|number> }

eg:

{ total : { $sum : "$quantity" } }

{ count : { $sum : 1 } }

↑

Simply ~~the~~ way to count no. of the docs in each group

Sum & group

```
db.persons.aggregate([{$group:{_id:"$age",
    count:{$sum:1}}}])
```

Sum, unwind, grp

```
db.persons.aggregate([{$unwind:"$tags"},
{$group:{_id:"$tags", count:{$sum:1}}}])
```

$Avg accumulator

- Calculates avg value of the certain values in the docs for each grp.

```
{$Avg:<expression>}
```

eg:

```
{$avg: {$avg:"$age"}}
```
(Age)

avg & group

```
db.persons.aggregate([{$group:{
_id:"$eyecolor", avgAge:{$avg:"$age"}}}])
```

result: 3 eyecolors with averageage.

Unary operators

$type	$or
$lt	$gt
$and	$multiply

In accumulators, the operators on the grp of the docs so they accumulate certain value. eg sum accumulates total count. But in unary, the operators perform operations for each doc. They dont work for entire grp. Thats y unary operators are usually used in the project stage

$type - It will take value of ~~some~~ certain field and op its be some type, like string or integer and so on. Such operators as '<', '>' and take conditions as a value. and retuns true or false value

$or - take values and if any of the value is true, then the result is true.

$multiply - multiply 2 or more value.

→ Unary operators are usually used in the $project stage

→ In the $group stage Unary operators can be

used only in conjunction with accumulators

$type

- Return BSON type of the fields value

  { $type : <expression> }

  BSON — no id and String Id

eg:

{ $type : "$age" }

{ $type : "$name" }

```
db.persons.aggregate([{ $project:
{name:1, eyecolorType:{ $type: "$eyecolor"},
 ageType:{ $type: "$age"}}}])
```

Result:

```
{_id : - - - -
"name": - - - - -
"eyeColorType" : "String"
"ageType" : "Int" }
```

$out stage

- writes resulting docs to the mongodb collection

  { $out : <outputCollectionName>" }

  eg:

  <$out : "new collection">

- $out must be last stage in the pipeline
- If o/p collection doesn't exist, it will be created automatically

eg:

```
db.persons.aggregate([{$group: {_id:
{age: "$age", eyeColor: "$eyecolor"}}},
{$out: "aggregationResults"}])
```

↓

docs from the $grp stage will be written to the collection aggregnResults

allowDiskUse : True

- All aggregatn stages can use max of 100MB of RAM
- Server will return error if RAM limit is exceeded
- following option will enable MongoDB to write stages data to the temporal files

  {allowDiskUse: true}

eg:

db.peeson.aggregate([], {allowDiskUse: true})

↓

tells server, that it can use tempores files instead of RAM.

$exists

{ field : { $exists : <boolean> } }

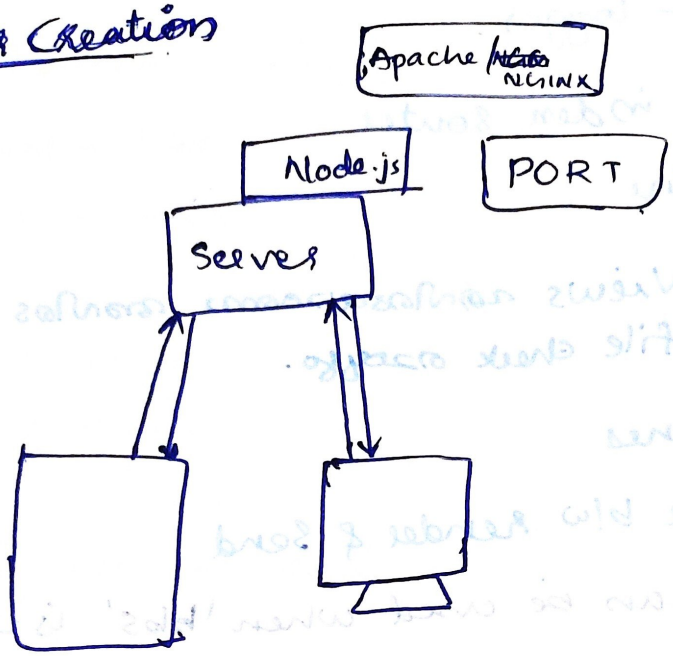
when boolean is 'true', $exists matches the documents that contain the field, including docs where the field value is null. If boolean is false, the query returns only the docs that do not contain the field.

db.persons.find({ qty : {$exists : true, $nin : [5,15] } })

This query will select all docs in the inventary collectn where the qty field exists and its value doesn't equal to 5 or 15

We can check the existance of the field in the specified collection/doc.

Server Creation



GET — data get ~~send~~

POST —

PUT —

DELETE —

We are using MVC — model view control

url — get method

Signup — Post method

→ app.use — middleware

```
app.use ((req, res, next) => {
    console.log('hello')
    next()
})
```

→ morgan - logger.

→ Routes - index routes

→ View engine

→ Render - views എന്നിടത്താണ് അതിനെ പോയി file check ചെയ്യും.

→ View engines

→ difference b/w render & send

render - can be used when 'hbs' is used.

Handlebars

→ hbs

→ Array il loop ചെയ്ത് work ചെയ്യാൻ

{{#each values}}

<h1> {{this}} <h1> (നമ്മൾ തരുന്നത്)

{{/each}}

Values ന്റെ ഉള്ളിലെ ഓരോ object നെയും കിട്ടും

Values = ["Rubeena", "Aswa", "Eshan", "Hessa"]

#each - for looping ന്റെ ഒരു statement

<h1>{{@index}}. {{this}} </h1>

result will be

0. Rubeena
1. Anwar
2. Eshan
3. Hesia

{{#if person.admin}}

<h1> This is admin </h1>

{{else}}

<h1> This is normal user </h1>

{{/if}}

const person =
{name: "Aysha",
Admin: true}

---

const person = {Name: "Rubeena", comments {comment : "Sample comment"}}

{{#with person}}

<h1>{{comments.comment}} </h1>

or

{{#with person}}
{{#with comments}}

<h1> {{comment}} </h1>
{{/with}}

This will display ...

— Layout hbs

we can add css, bootstrap sheet

= {{{ body }}} - page ൽ represent ചെയ്യാൻ

same style തന്നിരിക്കാം വേറെ pages ന്

{{ }} - variable represent ചെയ്യാൻ

mongodB

~~Sql~~

→ SQL - Structured query language

→ NO SQL - Not only " " "

→ CRUD

→ Collections : ഒരു section ന്റെ ഉള്ളിൽ multiple section കാണിക്കുന്നതിനെ collections എന്ന് പറയാം.

→ Collections create ചെയ്യാൻ - db.createCollection ('users')

→ db.users.find().Pretty() ⇒ code pretty ആക്കാൻ

{ "_____" : _____

"_____" : _____

"_____" : _____ }

→ To remove database

db.dropdatabase()

Mongodb + node.js

mongodb client connection:

```
MongoClient.connect('mongodb://localhost:
                    27017', (err, client) =>

    if(err)
     console.log('error')
    else
    (client.db('Project').collection('user').
                insertOne(req.body)

    })
```

Promise

→ CB fnctns to add two nos, with that result multiply by a no and finally the no should divide by some other no.

```
function add (num1, num2, callback) {
    callback(num1, num2)
```

```
function add (num1, num2, callback){
  let err = false
if (num == =0){
  err = true
}
callback (num1+num2, err)
}

add (10, 20, (sum, err) => {
  if(err){
    console.log ('first no. is zero')
  } else {
  console.log (sum)
  multiply (sum, sum, (product) => {
    console.log (product)
  division (sum, 10, division) => {
  console.log (division)
  }) }) } })
function multiply (num1, num2, callbac
  callback (num1 * num2)
}
function division (num1, num2, callback){
  Call back (num1/10), err)
```

This can be written in promise as:

```
const promise = require('Promise')
const { resolve, reject } = require('Promise')

function add (num1, num2) {
  return new promise((resolve, reject) => {
    if (num1 = 0){
    reject ("first number is zero")
    }
    resolve (num1 + num2)
  })
}
function multiply (num1, num2){
  return new promise (resolve, reject) => {
    if (num1 = 0){
  reject ("first number is zero")
    }
    resolve (num1 * num2)
  })
}
```

```
function div(num1, num2) {
  return new promise (resolve, reject) => {
    if (num == 0) {
  reject ("first no. is zero")
  }
  resolve (num1 / num2)
  })
  }

add (10, 20). then (sum) => {
  console. log (sum)
  return multiply (sum, sum)
}). then (Product) => {
  console. log (Product)
  return multiply (sum, Product, 10)
}). then (~~mult~~ div) => {
  console. log (division)
}) .
. catch ((err) => {
  console. log (err)
})
```

→ Poomise – 3 States

1. Pending State
2. Fullfill State (Resolve State)
3. Reject State

→ HBS[illegible] Loop [illegible] '# each' arrayname'