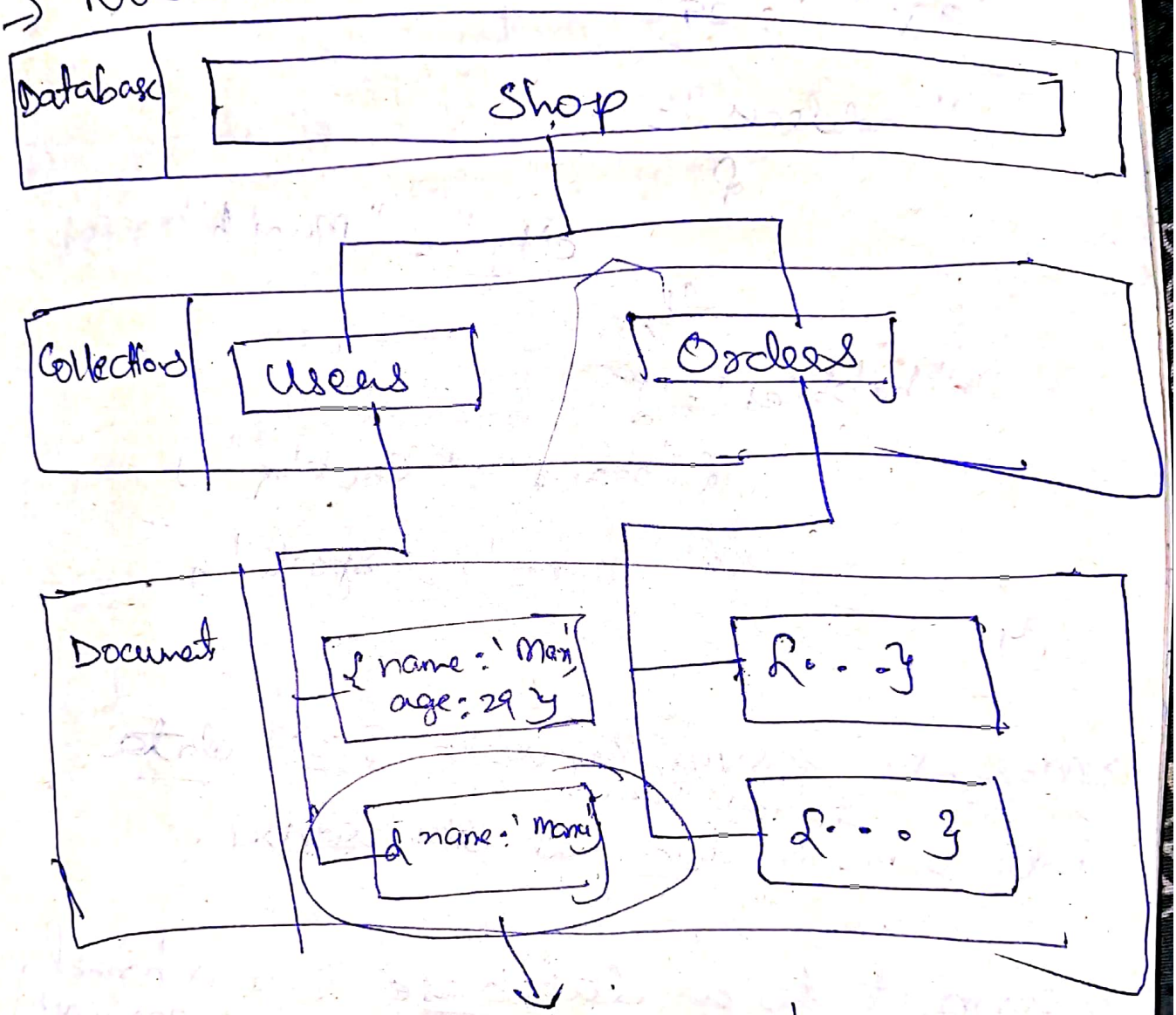
Bismilla - Hir - Rahman - Nir - Raheem

① mongoDB

→ sts a database-soln.

→ No schema.



Schemaless !

→ No / Few relations (increases speed & performance)

JSON data format

key → "name" : "max" ← value

```
{
  "name" : "max",
  "age" : 29,
  "address" :
      {
        "city" : "Munich"
      },
  "hobbies" : [
      {"name" : "cooking"},
      {"name" : "sport"}
  ]
}
```

→ MongoDB server converts JSON data into a BSON form at the server. (binary-JSON)

→ connect to a database (or a brand new db)

```
use shop
```
↳ db name

→ we can create a new collection :- (JSON object)

```
db.products.insertOne({"name": "max", "price": 29})
```

db → refers to database we are connected for

products → name of collection (doesn't need to exist, a new will be created)

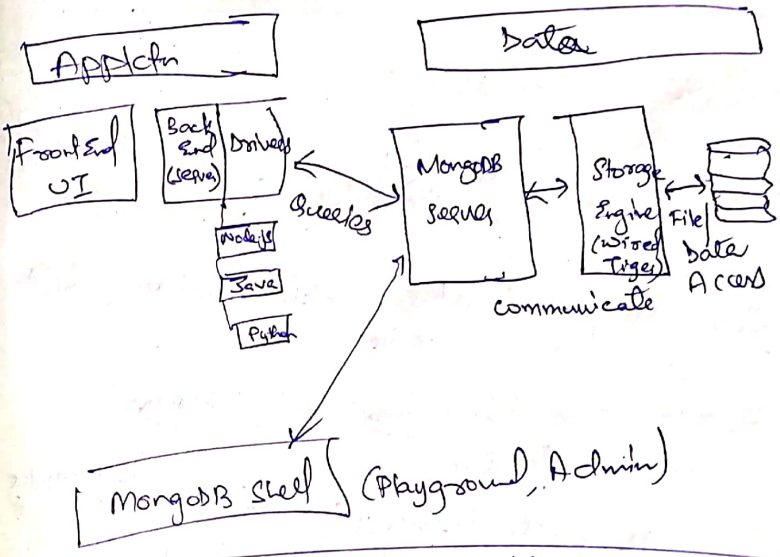
→ db.products.find()

↳ All docs in the collection is returned.

For ~~nested~~ embedded docs

```
db.products.insertOne({"name": "max",
"details": {"model": "Intel 8760"}})
```

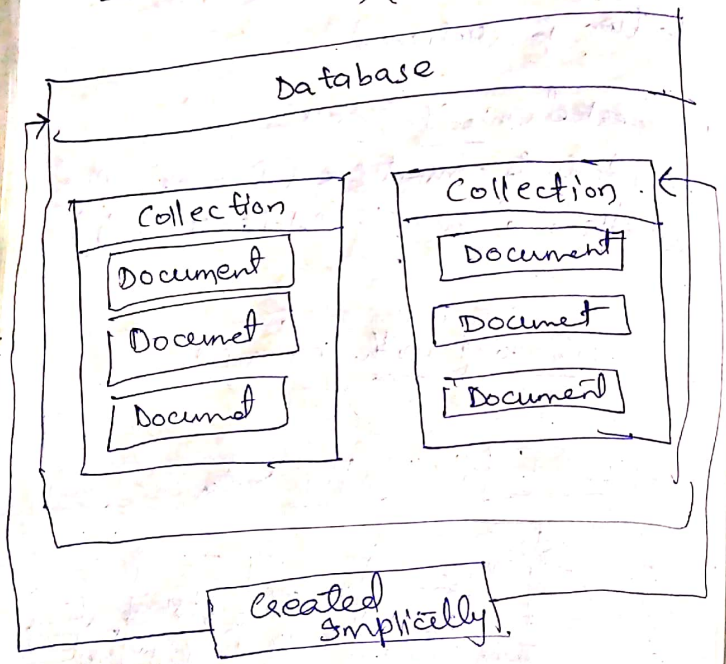
Working with mongoDB



→ Upon insertion, mongo DB provides a unique ID (object id)

Databases, Collections, Documents

↳ (like tables in sql)



showdbs → will return which databases exist in the server.

→ we can also have assign unique ids too:-

```
db.product.insertOne({"Name":"Milk",
                      "_id":"1x1-1x1"})
```

CRUD ops

(I) Create

```
insertOne(data, options)
insertMany(data, options)
```

(II) Read

```
find(filter, options)
findOne(filter, options)   // returns 1st matching doc.
```

(III) Update

```
updateOne(filter, data, options)   // filter: to filter docs to change; data: describing the change
updateMany(filter, data, options)
replaceOne(filter, data, options)   // replace the doc.
```

(IV) Delete

```
deleteOne(filter, options)
deleteMany(filter, options)
```

Ex:→ db.products.deleteOne({"name":"Maggi"})

→ db.products.updateOne({~~distance~~ "name":"Miyt"},
{$set:{"price":0}})

↳ // syntax keyword for update. Note:- If price doesn't exist, new entry will be made to collection

→ db.products.updateMany({}, {$set:{"price":1}})

* To update all docs without any criteria.

insertMany

db.products.insertMany([
{"name":"AB", "age":22},
{"name":"As", "age":28}])

// using [ ] (array) we can do this

find

db.products.find({"price":0}) ✓

→ Now to obtain docs having price greater than 0.

db.products.find({"price":{$gt:0}})

Difference between update() & updateMany()

→ They both update all the matching documents but update() → takes no $set, and replace the given doc with existing doc (i.e. existing id, except id)

db.products.update({"name":"Miyt"}, {"mftr":"MDH"})

→ To change a doc use replaceOne() instead

Understanding find() & cursor object.

find()
↓
Cursor object (it is created when 20 docs are found)
↓ Allows us to
Cycle through result

Projection

```
{
  "_id": "..."
  "name": "Max"
  "age": 29
  "job": "instructor"
}
```
→ In database.

Projection

But if the requirement in application is only

```
{ "name": ...
  "age": ... }
```

→ So, while transferring data to application we transmit more data over wires thereby increasing time & reducing efficiency.

→ This is where projections are helpful.

```
db.products.find({}, {"name": 1})
```

no filter, so all

Projection

"name" has 1. It means only "name" fields will be returned

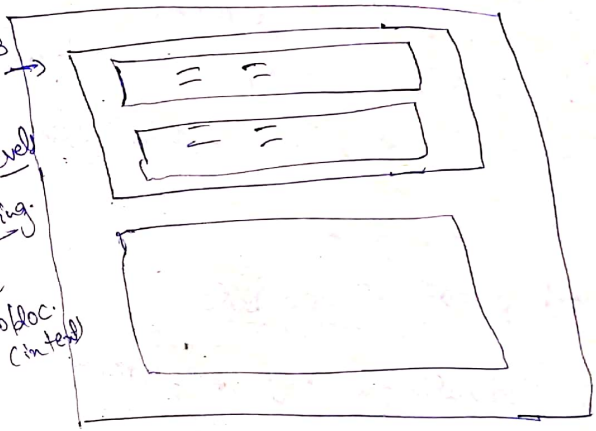
1 is default, to exclude a particular key, we have to explicitly mention 0. (i.e in case of id)

```
db.products.find({}, {"name": 1, "_id": 0})
```

Embedded Documents

(Doc. within a doc)

1) MongoDB allows upto 100 levels of nesting.
2) Max 16 mb/doc. (in text)



Arrays

1) Array of embedded doc.

2) Arrays can hold any data.

Ex:- of Embedded Doc (Doc. within a Doc)

```
db.products.updateMany({},{$set:
{"status":{"description":"Biscuit"
,"cityprod":"Delhi"}}})
```

Ex: of Arrays (we can provide multiple vals to key)

```
db.products.updateOne({"name":"Max"},
{$set:{"hobbies":["cooking","sleeping"]}})
```

Accessing Structured Data

→ `db.products.find({"name":"Max"}).hobbies`

(or)

`db.products.find({"hobbies":"cooking"})`

→ `db.products.find({"status.description":"Biscuit"})`

Resetting Your Database

① To get rid of database.

```
use dbname
db.dropDatabase()
```

② To get rid of a single collection in a database

```
db.myCollection.drop()
```

Storing your data correctly

① MongoDB enforces no schemas. Documents don't have to use the same schema inside of one collection.

Ex:-
```
-> use shop
-> db.products.insertOne({"name":
"A book","price":12.99})
-> db.products.insertOne({"title":
"T-Shirt","seller":{"name":"max",age:29}})
```

② To Schema or Not to Schema

Chaos! ▷ ▷ ▷ ▷ ▷ SQL World

Products

{ title: "Book", price: 12.99 }

very different!

{ name: "Bottle", available: true }

Products

{ "title": "Book", "price": 12.99 }

Extra Data

{ "title": "Bottle", "price": 5.99, "availability": true }

Products

{ "title": "Book", "price": 12.99 }

Full Equality

{ "title": "Book", "price": 6.03 }

Note:-

null ⇒ is a value we can assign to a field to tell that there is no value available.

db.products.insertOne({ "name": "Door", "details": null })

Data Types

i) Text (Ex:- "Max")

ii) Boolean (Ex:- true (or) false)

iii) Number

Integer (int 32 bit) (Ex:- 5)

NumberLong (int 64) 1000000000

NumberDecimal Ex:- 12.99

iv) ObjectId (Ex:- ObjectId("5f3ad"))

v) ISODate ------ Timestamp

Ex:- ISODate("2018-09-09")

Timestamp(11423532)

vi) Embedded Document

{ "a": { ... } }

vii) Array (List of values)

{ "b": [...] }

Ex:- use company

⇒ db.companies.insertOne({ "name": "Apple", "foundingdate": new Date(), "insertedat": new Timestamp() })

new Date() ✓ refers to current date

new Timestamp() ✓ internal BSON util (based on current time in ms)

Ex:- ⇒ db.stats() // provides info like dbname, no. of collections, no. of objects, avg. obj size, data size, etc.

Ex:- → db.numbers.insertOne({"a": 1})

By default, it is stored as 64 bit

(obj size : 33)

if we use :-

→ db.numbers.insertOne({"a": NumberInt(1)})

(we can store as 32 bit, thereby reducing obj size)

→ type of db.numbers.findOne().a

⇒ number

Understanding Relations

① Nested / Embedded docs.

Customes

```
{
  "name": "max",
  "age": 29,
  "address": {
      "street": "Second Avenue",
      "city": "New York"
  }
}
```

② References

customers

```
{
  "name": "max",
  "favBooks": ["id1", "id2"]
}
```

Books

```
{
  "_id": "id1",
  "name": "Lord of the Rings 1"
}
```

One to One Relation (Reference)

Patient A ⟷ Summary A

"One patient has one disease summary & a disease summary belongs to one patient"

Reference approach

→ db.patient.insertOne({"name": "max", "age": 29, "disease summary": "sm1"})

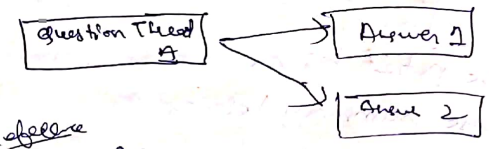
→ db.summaries.insertOne({"_id": "sm1", "diseases": ["cold", "broke leg"]})

• we have to obtain disease summary of a patient

```
var a = db.patient.find({"name": "max"}).diseaseSummary
// a will have "sm1" stored.
then:
db.summaries.find({"_id": a})
```

→ But this method is inefficient, so we shall use Embedded doc during strong one-to-one relation

One to Many Relation



Ex:- Refference approach :

→ db.question.insertOne({"creator": "mikey", "question": "How does CPU work", "answers": ["a1", "a2"]})

→ db.answers.insertMany([{"_id": "a1", "text": "Done"}, {"_id": "a2", "text": "Done with stem root"}])

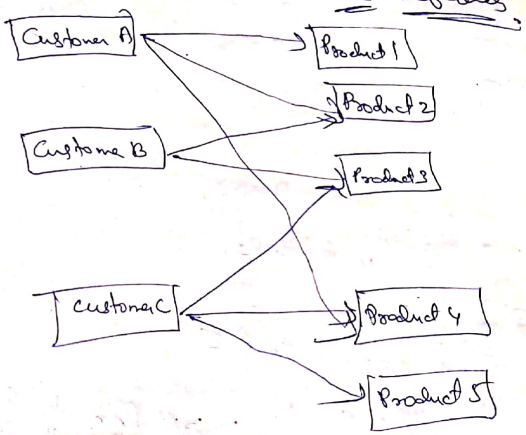
→ var a = db.question.findOne().answers

→ db.answers.find({"_id": a[0]})

Ex. of embedded approach :- (use this for efficiency)

db.question.insertOne({"creator": "mikey", "question": "How does CPU work", "answers": [{"text": "Done"}, {"text": "Done with stem root"}]})

Many To Many we Refference



"One customer has many products, a product belongs to many customers"

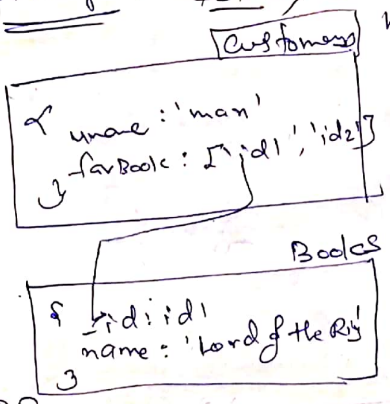
→ we model many to many often with refferences.

Relation - Options

Embedded Doc	Refferences
→ Great for data that belongs together & is not really overlapping with other data.	→ split data across collections. → Allows you to overcome nesting & size limits.

$lookup()

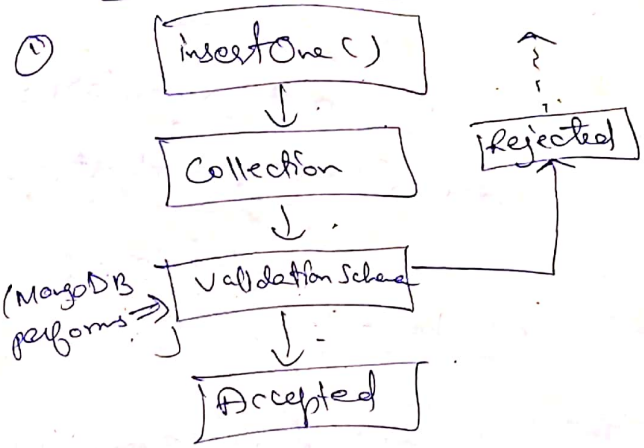
(A helpful tool that allows us to fetch, 2 related docs merged together in 1 doc in 1 step)




```
db.customers.aggregate([
  { $lookup: {
      from: "books",          // from which other collection you want to relate the doc.
      localField: "favBook",  // On the collection we are running query, where can we find the reference
      foreignField: "_id",    // which field are you relating in target collection
      as: "favBookData"       // alias under which it will be merged.
  } }
])
```

→ It costs performance, so if possible use embedded doc.

Schema validation



2 Steps involved

validationLevel	validationAction
1) Which docs got validated?	1) What happens if validation fails?
a) strict (All inserts & updates checked)	a) error (Throw error & deny insert/update)
b) moderate (all inserts are checked, but updates are checked only if docs were valid before)	b) warn (Log warning but proceed)

Adding Collection Document Validation

Name of collection

db.createCollection("posts", {validator:

{$jsonSchema: {bsonType: "object",

everything added to

collection must be object,

properties: {

//how every field must be.

name: {bsonType: "string"},

required: ["name", "age"]

fields which are absolutely req'd.

age: {bsonType: "number" } } } })

Changing the validation action

→ To change the validation, we need

db.runCommand({collMod: "posts", pass

to run the query as admin.

the validator from above, validationAction: "warn"})

Ex:- validator: {$json ---- ---

Summary

1) Schemas should be modelled based on application needs.
2) Imp factors are Read & write frequency, relations, amount & size of data.

---

note

→ help (gives a list of commands)
→ db.help() (Commands for db)
→ db.collectionName.help() (Commands for collections)

MongoDB Compass

→ Is a GUI to manage a database. (CRUD, etc)
→ We will connect to a local host
→ We can perform all the operations in Compass.
→ In Enterprise Compass, we have a special option of Schema.

CREATE DOCUMENTS

a) insertOne()

db.collname.insertOne({"field":"value"})

b) insertMany() // we can insert 1 doc too but within [.....]

db.collname.insertMany([{"field":"value"}, {"field1":"value1"}])

c) insert()

db.collname.insert( )

↓

we can use for both one & many doc insertion.

→ This is not recommended as it doesn't return inserted id's.

Providing our own id:-

→ db.hobbies.insertMany([{"_id":"sport", "name":"sport"}, {"_id":"car", "name":"car"}])

→ We have our own unique ids & thereby unique objects ids aren't created by server.

orderedInsert:- Every element you insert is processed standalone, but if one fails, it cancels the entire insert op but does not roll back.

Ex:- db.hobbies.insertMany([{...}, {...}, {...}, {...}])

↓ If this insert is successful.

→ and this fails due to some reason further insert won't proceed, but previous insert will be recorded.

→ To change the behaviour, we will pass another argument to configure

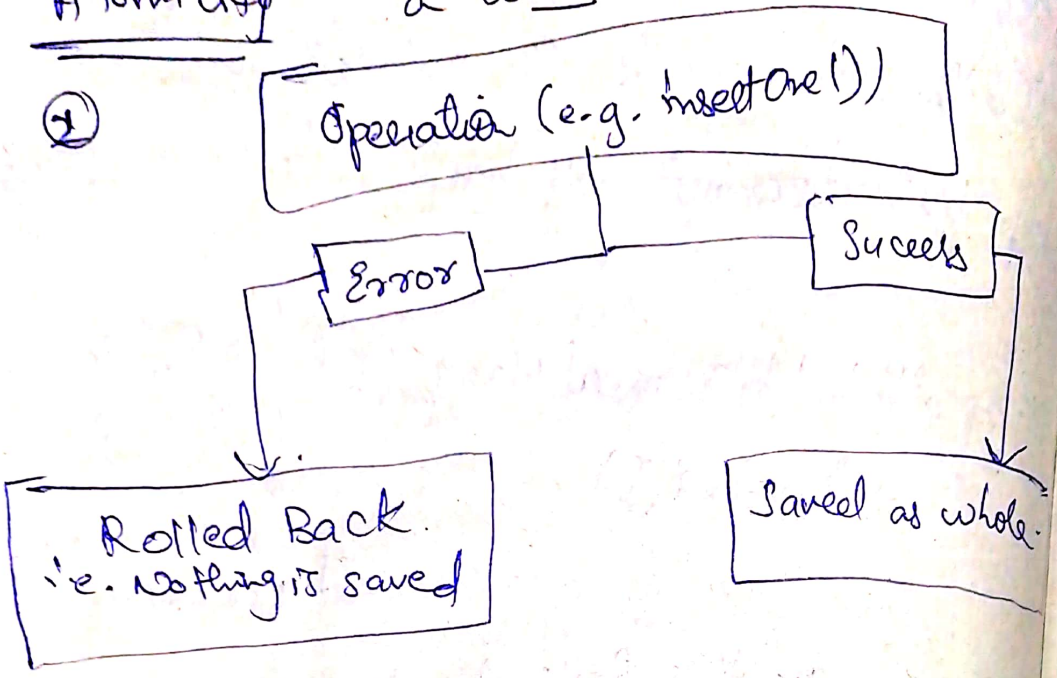
db.hobbies.insertMany([{...}, {...}, {...}], {"ordered": false})

→ Due to this the insertMany op. will proceed further, even if 1 of the insert fails, but rollback sth.. all prev. inserts will be recorded though.

Write Concern

Atomicity ① (Transaction succeeds as a whole or fails as a whole)

②



③ MongoDB CRUD ops are Atomic on the Document Level.

i.e

```
db.hobby.insertOne({"nae": "A",
    "age": 21, "text": "th--- --- "})
```

↙

If due to some reason this fails, entire name & age won't be recorded too.

④ In case of insertMany(), atomicity is checked at document level individually

```
db.hobby.insertMany([{...}, {--- }, {...}])
```

↙

individual atomicity is checked.

Importing Data

1) open cmd.
2) Navigate to place where .json file is located. (tv-shows.json) using cd.
3) mongoimport tv-shows.json -d movieData (db name)
-c movies --jsonArray --drop
(file name) (coll. name) (if coll. already exist, it will be dropped & recreated)

(To tell that json file consists of multiple docs)

Methods, Filters & Operators

Access mongo database: db . Access this collection: myCollection . Apply this method: find(Filter: equality / Single value {"age": 32})
Field: "age" value: 32

Range Filters

db.myCollection.find({"age": {$gt: 30}})
Field: "age" operator: $gt value: 30

Read Operators

1) Query & projection
   a) Query selectors
   b) Projection Operators

Update Operators
a) fields b) Arrays.

Query Selectors have diffferent operators like :-
a) Comparison b) Logical c) Element
d) Evaluation e) Array f) Comments
g) Geo spatial

Projection Operators
a) $ b) $elemMatch c) $meta
d) $slice.

Comparison operators → $eq, $gt, $gte, $in (matches any of the values specified in an array), $lt, $lte, $ne (not equal to specified value), $nin (matches none of values specified in an array)

Eg:-
→ db.movies.find({"runtime": {$eq: 60}})
That's similar to :-
db.movies.find({"runtime": 60})

→ db.movies.find({"runtime": {$ne: 60}})
(Similarly consider all operators)

→ db.movies.find({"runtime": {$in: [30, 42]}})
// returns a doc where runtime is 30 (or) 42

→ db.movies.find({"runtin": {$nin: [30, 42, 57]}})

Logical operator

$and, $not, $nor, $or

$and → returns doc that match both clauses
$not → returns doc that match cndtn of both clauses.
$nor → returns doc that fail to match both clauses
$or → returns doc that match cndtn of either clause.

→ db.movies.find({"rating": {
// find doc with rating lower than 5 (or) greater than 10.6
→ db.movies.find({$or: [{"rating": {$lt: 5}}, {"rating": {$gt: 10.6}}]})
→ similarly write nor query

$and (Joins queries & returns doc that match both clauses)

db.movies.find({$and: [{"rating": {$gt: 5}}, {"genre": "drama"}]})
(or)
db.movies.find({"rating": {$gt: 5}, "genre": "indi...

$not (inverts effect of query expression & returns docs that do not match query expression)
→ not commonly used.

Element operators

a) $exists → (matches docs that have specified field).

b) $type → (selects docs, if the field is of specified type)

Ex:- db.users.find({"age": {$exists: true}})

db.users.find({"phone": {$type: "number"}})

Evaluation operators

Name	Description
1) $expr	→ Allows use of aggregation expression within query language.
2) $jsonSchema	→ validate docs against JSON schema.
3) $mod	→ Performs modulo op. at value of a field & selects docs with a specified result.
4) $regex	→ selects docs where values match a specified RE.
5) $text	→ Performs text search
6) $where	→ matches docs that satisfy a JS expression.

Ex:- (Pattern: /.../)

db.movies.find({$summary: {$regex: /musical/}})

→ W.A.Q where volume > target

db.sales.find({$expr: {$gt: ["$volume", "$target"]}})

Array

1) $all 2) $elemMatch 3) $size

→ (matches arrays that contain all elements specified in query)

→ (selects docs if the array field is of a specified size)

Ex:-

db.users.find({"hobbies": {"$size": 3}})

provide exact no.

(3 refers to number of entries in array i.e 3)

ex: hobbies: ["A", "B", "C"]

Ex:-

db.movieStats.find({$genre: {$all: ["action", "thriller"]}})

-P.T.O

$elemMatch (selects docs if elements in the array field matches all the specified $elemMatch conditions)

Ex:- title should be sports & freq greater than 33

```
db.users.find({"hobbies": {$elemMatch: {"title": "sports", "frequency": {$gte: 33}}}})
```

i.e it means title & frequency belongs to same doc within array.

Applying Cursors

```
-> const dataCursor = db.movies.find()
-> dataCursor.next() // obtains next doc i.e after 20th doc.
-> dataCursor.next()
-> dataCursor.forEach( doc => { printjson(doc) })
```

↓

It takes a func & is executed on every element (doc) fetched via cursor

```
-> dataCursor.hasNext() // true or false.
```

Sorting results

```
-> db.movies.find().sort( { "rating.average": 1 } )
```

(1 is for ascending, -1 for descending)

we can have multiple fields also.

```
.sort( { "rating": 1, "runtime": -1 } )
```

Skipping Results

(on a page if we are having 10 results, then while on Pg 2, we have to skip first 10 results).

```
db.movies.find().sort( { "rating": 1 } ).skip(10).limit(10)
```

↳ (allows us to retrieve only a particular no. of elements in a query)

Note:- In whichever order you write query, MongoDB will first sort, then skip and then Limit.

Slice operator

UPDATE OPERATIONS

Field update operator

Name	Description
1) $currentDate	→ sets the value of field to current date, either as Date or Timestamp.
2) $inc	→ increment value of field by specified amount
3) $min	→ only updates field, if specified value is less than existing field value
4) $max	→ ... greater ...
5) $mul	→ multiplies value of field by specified amount.
6) $rename	→ Renames a field
7) $set	→ sets the value of field in a doc.
8) $setOnInsert	
9) $unset	→ Removes the specified field from a doc.

Ex:- db.persons.updateOne({"name": "Mike"}, {$set: {hobbies: [{"title": "sports", "frequency": 5}]}})

⇒ This field will be overridden.
⇒ All existing fields are untouched.

Updating multiple fields with set

→ db.users.update({"name": "max"}, {$set: {"age": 40, "phone": "8343"}})

Incrementing & Decrementing values:-

→ db.users.updateOne({"name": "Manuel"}, {$inc: {"age": 1}})
// increases value by 1.

Ex:- age: -1 // decrement

⇒ We can use inc & set in 1 update query.

→ db.users.updateOne({"name": "Manuel"}, {$inc: {"age": 1}, $set: {"number": 30}})

Note:- We can't use same field for both inc & set as it will create conflict.

UPDATE OPERATORS

Using min, max, & mul

→ db.users.updateOne({"_id":1},
{$max:{"highscore":950}})

(The highscore existing value is 800
the above op uses $max to compare
800 and specified value 950 &
update the value of highscore to
950, since 950 is > 800)

→ db.users.updateOne({"_id":1},
{$min:{"highscore":60}})

→ db.users.updateOne({"_id":1},
{$mul:{"age":3}})
// multiplies the
field value by 3.

Getting Rid of field

→ db.users.updateOne({"name":"Mikey"},
{$unset:{"highscore":""}})
(default value
doesn't)

Renaming a field

→ db.users.updateOne({"name":"Mikey"},
{$rename:{"highscore":"manScore"}})

Updating upsert()

→ we want to update a doc where
we are not sure if it exists (or) not.

→ db.users.updateOne({"name":"Mik"},
{$set:{"age":19}}, {upsert:true})

// if doc. doesn't exist,
it will be created.

Updating Matched Array Elements

("hobbies": [{ ... }, { ... }] — db this is matched, only that ele is updated)

Ex:-

→ db.users.updateMany({"hobbies": {$elemMatch: {"title": "Sports", "frequency": {$gte: 3}}}}, {$set: {"hobbies.$.highFrequency": true}})

- `$` ↓ automatically refer to ele. (array ele) matched in query
- highFrequency ↳ New field creating

Updating all array elements

"hobbies": [{ ... }, { ... }]

If a match is found, we are updating each array ele.

→ db.users.updateMany({"totalage": {$gt: 30}}, {$inc: {"hobbies.$[].frequency": -1}})

- `$[]` ↓ selecting all array element, when a match is found

Finding & Updating specific field in array

Adding elements from array

→ db.users.updateOne({"name": "Maria"}, {$push: {"hobbies": [{"title": "book", "frequency": 5}]}})

- hobbies ↳ array in which to insert
- `$push` ↓ It does not override the array (like $set), but instead adds another field

→ To insert many docs at a time in array :-

... {"hobbies": {$each: [{title: ...}, {title ...}]}})

Removing elements from array

→ db.users.updateOne({"name": "Uei"}, {$pop: {"hobbies": {"title": "Hiking"}}})

- hobbies ↓ array from which ele is to be removed
- Remove ele. where title is hiking

→ Remove last ele from array :-

```
db.users.updateOne({"name": "chris"}
  {$pop: {"hobbies": 1}})
```

↳ (To remove last ele
(-1 → To remove 1st ele)

$addToSet (same as push, but wont allow multiple insertion using $each)

```
db.users.updateOne({"name": "Mikey"}
  {$addToSet: {"hobbies": {"title": "Hiking", "frequency": 2}}})
```

(hobbies is array)

→ push allows insertion of duplicate values.

→ addToSet allows only unique values.

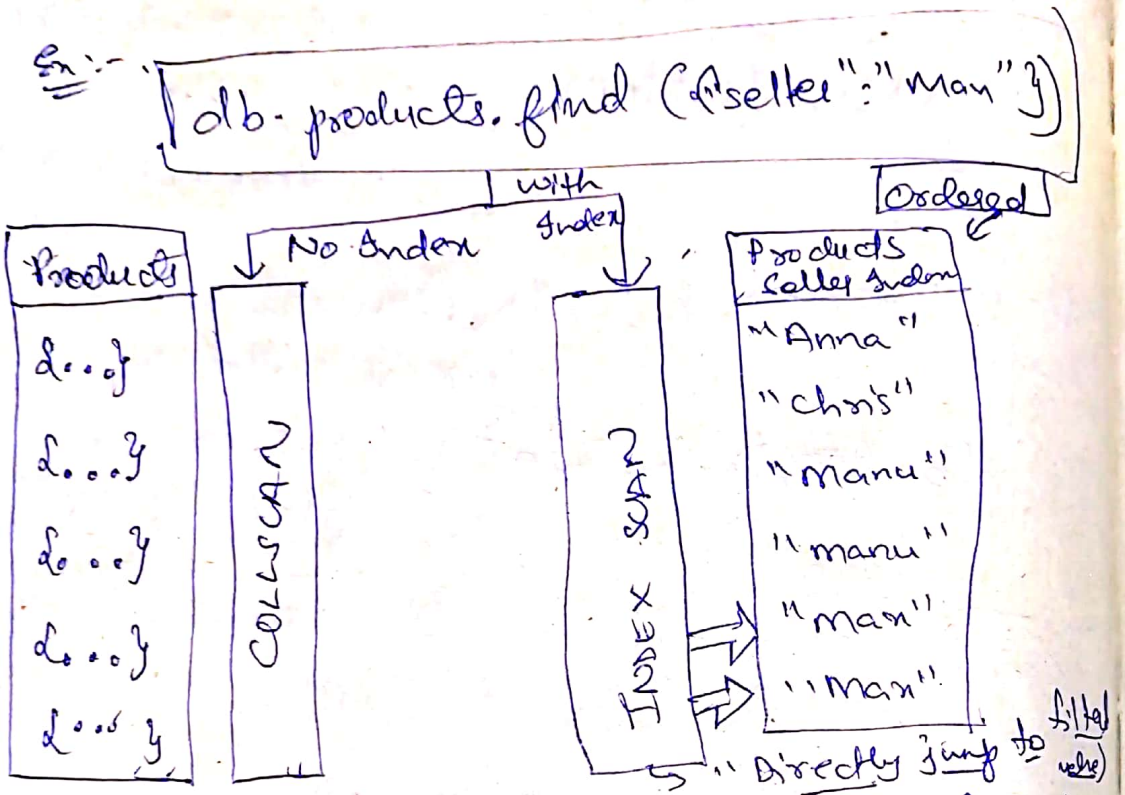
Document Deletion Methods

1) `db.users.deleteOne({"name": "chris"})` (deletes 1st record)
2) `db.users.deleteMany({"age": {$gt: 30}})`
3) `db.users.deleteMany({})` // All docs are deleted from collection
4) `db.users.drop()` // Deletes the 'users' collection.
5) `db.dropDatabase()` // Deletes the database itself

Indexes (Retrieving Data Efficiently)

① Index can speed up Find, Update & Delete queries.

Ex:- `db.products.find({"seller":"Max"})`



→ When there is no index, MongoDB goes through entire collection, look at every doc. & see if seller = Max. (and is time consuming)

→ Index is an ordered list of all the values, that are placed (or) stored in the seller key.

→ Every item in the index has the pointer to the full document it belongs to ; which allows MongoDB to perform

Index Scan :

Note :- Use of too many indexes causes performance cost in Insert operation, it may speed up Find operation, but it is not worth.

Adding a single field Index

Note :- To determine how mongo DB is processing the output of a query :-

→ db.contacts.explain().find({"age": {$gt: 60}})

(It provides details like winning Plan etc)

To obtain specific details :-

→ db.contacts.explain("executionStats").find({"age": {$gt: 60}})

(It provides details like executionTimeMillis, nReturned (No. of docs returned), etc.

Creating Index :-

db.contacts.createIndex({"age": 1})

(-1 → sort in desc. order)

values stored in asc. order

Field on which index is created

Note:- It reduces execution TimeMill is _ and two execution stages are "IXSCAN". & "FETCH" (takes ptrs & reach out to actual colls & fetch real docs)

(It returns keys with pointers to docs)

Note:- Index is a simple list of values + pointers to original doc:-

for ('age' field)

(29, "address in memory")

→ Since values are sorted, the the process becomes quicker to search, because you can skip the ranges.

Understanding Index Restrictions

Note: To drop an index use:-

```
db.contacts.dropIndex({"age": 1})
```

→ If you have a query, that will return majority of docs, then index will be slower.

Creating Compound Index

~~→ Text index:- index on a text field.~~

```
db.contacts.createIndex({"gender": 1})
```

~~drop~~ db.contacts.dropIndex({"gender": 1})

⇒ compound index

```
db.contacts.createIndex({"age": 1, "gender": 1})
```

- The order is important, because a compound index (more than 2 fields) will store a 1 index where each value is combined value.)

→ The index will work for all (age & gender combined fields) and only for (age fields) (left most). For only gender fields, index isn't deployed.

Using Indexes for sorting

→ After creating index on age & gender :-

```
db.contacts.find({"age":35}).
        sort({"gender":1})
```

→ This Indexes while sorting is very important as MongoDB has threshold of 32MB to fetch & sort documents. In case of large no. of documents, it may lead to Time Out.

Understanding Default Indexes

→ db.contacts.getIndexes() (returns all indexes in a collection)

→ MongoDB has a default index on "_id" field.

Configuring Indexes

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

↙ configuring index (i.e email field should contain unique val)

Understanding Partial Filters

→ ~~Certain values with~~

Ex:-

```
db.contacts.createIndex({"age":1},
{"partialFilterExpression":{"gender":
                        "male"}})
```

// It creates an index for age field, & it will store only indexes for gender male

// $gt, $lt, etc are also supported.

Ex:-

```
db.contacts.createIndex({"age":1},
{"partialFilterExpression":{"age":{$gt:60}}})
```

Time To Live Index

→ useful for self-destroying ~~data~~ (like session)

Ex:-

→ db.customer.insertOne({"data":"acidfist",
"createdAt":new Date()})

→ db.Customer.createIndex({"createdAt":1})

↳ we can configure further :-

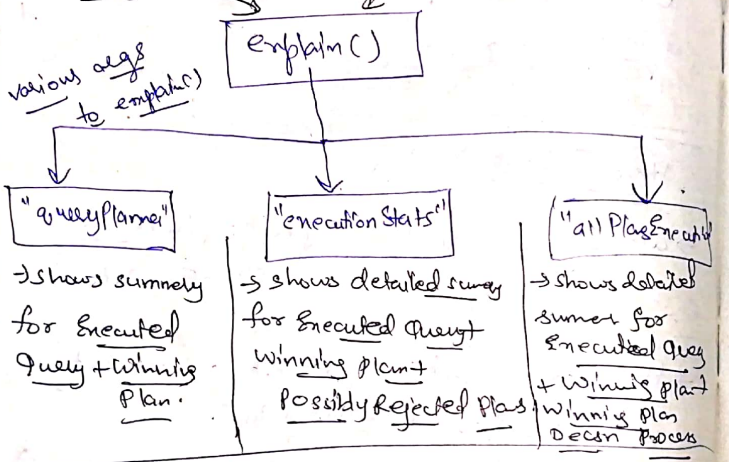
db.customer.createIndex({"createdAt":1}

{"expireAfterSeconds": 10})

TTL works on date fields, which means every ele shud be removed (deleted) after 10 seconds.

Ex:- used in online shop cart, where data shud clean up itself, you dont need to write complex scripts, instead "expireAfterSeconds" works.

Query Diagnosis & Query Planning



Efficient Queries & Covered Queries

Millisecond Processing Time ← look at

IXSCAN (Index scan) typically beats COLLSCAN

Shd be as close as possible, or # of docs shud be 0.

1) No. of keys (in index) examined
2) " " Docs examined
3) No. of Docs returned

Shud be as close as possible (or) no. of docs shud be 0.

Covered Query: using projections to return only the indexed field. (If doesn't have to examine docs)

How MongoDB Rejects a Plan?

→ db.customers.createIndex({name: 1})
→ db.customers.createIndex({age: 1, name: 1})

→ The fastest to reach 100 docs is Winning Plan
→ The winning plan is cached for a particular query, & in future if is used for similar query.
→ The winning plan is cleared from cache when. a) index is rebuilt
b) other indexes are added (or) removed.
c) MongoDB server is restarted

Using MultiKey Indexes

→ db.contacts.insertOne ({name: "Max", hobbies: ["Cooking", "Sports"], addresses: [{street: "Main Street"}, {street: "Second Street"}]})

→ db.contacts.createIndex ({hobbies: 1})

→ db.contacts.find ({hobbies: "Sports"})

// MongoDB treats this as MultiKey, since it is index on array of values

→ They are stored differently

→ db.contacts.createIndex ({"addresses.street": 1})

→ db.contacts.find ({"addresses.street": "Main Street"})

The index is not used, MongoDB stores index as doc, & it doesn't build field of docs.

→ If index is placed on "addresses.street" index is used.

Understanding "text" Indexes

→ (special multi-key index)

→ An Text index is a special type of index supported by MongoDB which turns text into an array of single words it removes stop words (is, that, a, the)

→ It creates an array of keywords.

product | must | buy | fans | modern | fiction

(This product is a must-buy for all fans of modern fiction!)

→ db.products.insertMany ([{title: "A Book", description: "This is an awesome book about a young artist!"}, {title: "Red T-Shirt", description: "This T-shirt is red & it's pretty awesome!"}])

// using text index

→ db.products.createIndex ({description: "text"})

Keyword "text" index.

Note:- You only have almost 1 text index as its expensive.

→ db.products.find({$text: {$search: "awesome"}})

⇒ we can also search for a specific phrase by wrapping in double quotes

⇒ db.products.find({$text: {$search: "\"red book\""}})

→ This is much faster than Regular expression.

Text Indexing & Sorting

db.products.find({$text: {$search: "awesome t-shirt"}}).sort({score: {$meta: "textScore"}}).pretty()

→ Docs will be sorted based on textScore (desc. order)

Creating Combined Text Indexes

→ we can't have multiple 2 text indexes so we use combination

→ Deleting text index is bit different.

→ db.products.createIndex({title: "text", description: "text"}) // There will be 1 text index but it contains keywords for title & description

→ db.products.find({$text: {$search: "shirt"}})

Using Text Indexes to Exclude Words

→ db.products.find({$text: {$search: "awesome -t-shirt"}})

↓

⊖ It indicates t-shirt should exist

Setting Default Language & using Weights

To drop a text index (this is different than normal index)

db.products.dropIndex("index name")

↖ get this from db.product.getIndexes()

→ db.products.createIndex({title: "text", description: "text"}, {default_language: "german"})

↙

configures (It will define how words are stemmed (like ing are removed) & what stopped words are removed (i.e.))

→ We can define different weights for different fields.

→ weights become important, when MongoDB calculates scores.

db.products.createIndex({title:"text", description:"text"}, {default_language: "english", weights: {title:1, description:10}})

→ It means description will weigh 10 times more than title.

→ We can obtain scores for a query

```
db.products.find({$text:{$search:"red"}}, {score:{$meta:"textScore"}})
```

Building Indexes

Working with GeoSpatial Data

Adding GeoJSON data:

→ Open a location in google map & obtain location from URL (..... latitude, longitude) ↓ ↓ are horizontal lines

<field>: { type: <GeoJSON type>, coordinates: [x, y] }

longitude ↘ latitude

Ex:-

→ db.places.insertOne({name:"AXIS BANK", location:{type:"Point", coordinates:[-122.476, 37.767]}})

→ To fetch w.r.t to find places near current location

→ db.places.find({location:{$near:{$geometry:{type:"Point", coordinates:[-122.471, 37.77]}}}})

(geoJSON data)

→ First add geospatial index

db.places.createIndex({location:"2dsphere"})

→ We can also provide $maxDistance & $minDistance args in order to define $near.

db.places.find({location: {$near: {$geometry: {type: "Point", coordinates: [-122.47, 37.77]}}, $maxDistance: 30 (in meters), $minDistance: 10 (in meters)}})

Adding Additional locations (using insertOne and do that)

Finding Places inside a certain area

→ const p1 = [-122.45, 37.77]

→ const p2 = [-123.10, 37.91]

→ const p3 = [-123.5, 36.99]

→ const p4 = [-122.76, 37.83]

→ db.places.find({location: {$geoWithin: {$geometry: {type: "Polygon", coordinates: [[p1, p2, p3, p4, p1]]}}}})

(helps us find all elements within certain shape)

Polygon must end with same coord

Finding out if a user is in a specific area

→ db.places.insertOne({name: "Golden Gate", location: {type: "Polygon", coordinates: [[p1, p2, p3, p4, p1]]}})

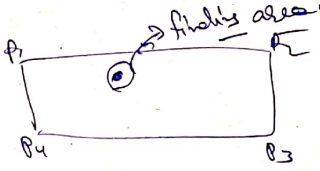
→ Now, creating index on location

db.places.createIndex({area: "2dsphere"})

→ Now finding out if a user is in a specific area:-

db.areas.find({area: {$geoIntersects: {$geometry: {type: "Point", coordinates: [-122.49, 37.76]}}}})

(location) (it is used on area field) (it returns all locations that have a point existing in them)



Finding Places within a certain radius

→ db.places.find({ location : { $geoWithin : { $centerSphere : [[-122.67..., 37.77], 1/6378.1] } } })

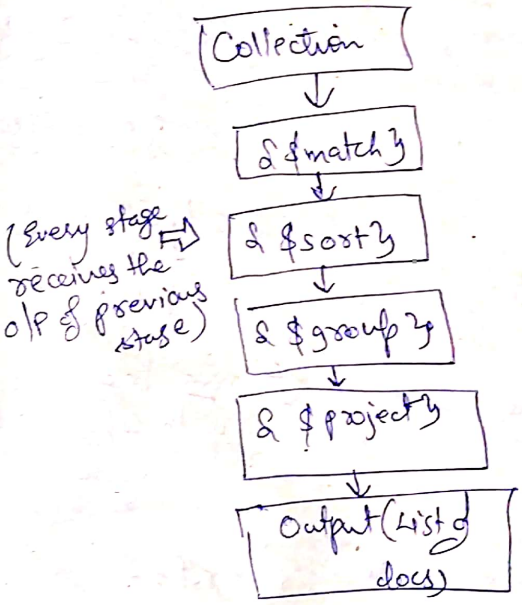
- $centerSphere : operator that allows us to get circle around a point.
- [-122.67..., 37.77] : Coordinates of center
- 1/6378.1 : Radius in radians (1 is km for 5 km we can write: 5/6378.1)

Note :- Always place index on field (working on geospatial data) cause some operations like $near require it.

AGGREGATION FRAMEWORK

(Retrieving Data efficiently & in a Structured Way)

① Building up a Pipeline of steps that runs on the data i.e retrieve data from collection & give output in the way you want.



② It is an alternative to find

→ db.persons.aggregate([{$match: {"gender": "female"}}, {$group: {"_id": "$location.state", totalPersons: {$sum: 1}}}])

- It takes an array because it defines a series of steps to be performed on data.

"$location.state" → refers to current doc.

→ acts as a filter

_id: syntax used to define by which value we want to group

{$sum: 1} → To add 1 for every doc grouped together

It allows us to group the data by certain field

$group creates o/p as:-

id: state	total Person
Yukoto	3
Malita	2

using sort

→ db.persons.aggregate([{$match: {"gender": "female"}}, {$group: {"_id": "$location.state", totalPersons: {$sum: 1}}}, {$sort: {totalPersons: -1}}])

$project (It works same as projection in find)

→ db.persons.aggregate([{$project: {_id: 0, gender: 1, fullName: {$concat: ["$name.first", " ", "$name.last"]}}}])

→ To make sure, first name begin with caps (first character should be capital)

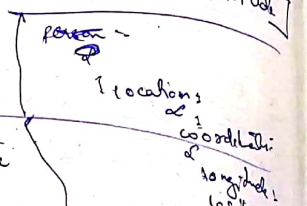
⇒ db.persons.aggregate([{$project: {_id: 0, gender: 1, fullName: {$concat: [{$toUpper: {$substrCP: ["$name.first", 0, 1]}}, {$substrCP: ["$name.first", 1, {$subtract: [{$strLenCP: "$name.first"}, 1]}]}, " ", "$name.last"]}}}])

(Index to obtain char. from the index; No. of character after the index)

⇒ gender: male fullName: ...

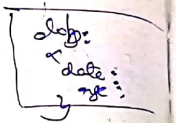
Turning the location into a geoJSON object

→ db.persons.aggregate([{ $project: { "_id": 0, "name": 1, "email": 1, "location": { "type": "Point", "coordinates": ["$location.coordinates.longitude", "$location.coordinates.latitude"] } } }])



Transforming the birthdate

→ db.persons.aggregate([{ $project: { "_id": 0, "name": 1, "birthdate": { $convert: { "input": "$dob.date", "to": "date" } }, "age": "$dob.age" } }])



$convert: { input: "", to: "" }

// used to convert from 1 data type to another, input takes the field to be converted to
to refers to the type to be converted to.
// additional parameters are onError: value & onNull: value

Using shortcuts for transformation

→ Instead of using $convert we can use specific operators for transformation like $toDate, etc.

"birthdate": { $toDate: "$dob.date" }

understanding $isoYear

$isoWeekYear (retrieves year out of date)

→ db.persons.aggregate([{ $group: { "_id": { "birthYear": { $isoWeekYear: "$birthdate" } }, numPersons: { $sum: 1 } } }])

$group (vs) $project

→ used for grouping multiple docs into 1 doc (n:1)

→ we use it to perform: Sum, Count, Average, Build Arrays

→ uses 1 doc & returns 1 doc. (1:1)

→ we perform Include/Exclude Transform fields (within a single doc)

Pushing elements into newly created Array

→ db.persons.aggregate([{$group: {_id: {age: "$age"}, allHobbies: {$push: "$hobbies"}}}])

$unwind :- Takes 1 doc & spits out multiple docs.
Ex:- db.persons.aggregate([{$unwind: "$hobbies"}])

Eliminating Duplicate values

→ use $addToSet instead of $push.

~~db.persons.ag~~

Using Projection in arrays ?

Getting length of an array :-

→ db.persons.aggregate([{$project: {_id: 0, numScores: {$size: "examScores"}}}])

↓ Calculates size of array

Using filter operator

↳ (allows us to filter arrays inside docs, inside a projection phase)

→ db.friends.aggregate([{$project: {_id: 0, scores: {$filter: {input: '$examScores', as: 'sc', cond: {$gt: ["$$sc.score", 60]}}}}}])

(as: assigned alias)

$$ ↓ syntax due to its reference to as field

Understanding $bucket (allows you to o/p data in buckets, for which we can calculate summary statistics)

→ It takes group by ~~field~~ parameter, where we define, by which field we want to define data into buckets.

→ db.persons.aggregate([{$bucket: {groupBy: "age", boundaries: [0, 18, 30, 50], output: {numPersons: {$sum: 1}}}}])

("age" ↳ is the group key; 0,18 = 1st bucket, 18,30 = 2nd bucket, 30,50 = 3rd bucket; numPersons ↳ no. of persons in a bucket)

o/p (0 to 18 is a bucket)
_id : 18, numPersons : 868
_id : 30, " " : 1609

Writing Pipeline results into a new collection

```
db.persons.aggregate([ { ... }, { ... }, { $out: "transformed..." } ])
```

$out → To write pipeline result

"transformed..." → collection name, it can be existing one or new is created

Working with Numeric Data

→ We can work with these 4 types of numbers:-

Integers (Int 32 bit)	Longs (Int 64 bit)	Doubles (64 bit)	High Precision Doubles (128 bit)
→ Only full numbers	→ Only full numbers	→ Numbers with decimal places	→ Numbers with decimal places
→ Range: -2,147,483,648 to 2,147,483,647	Range: -9,223,372,036,854,775,808 to 807	→ Decimal values are approximated	→ Decimal values are stored with high precision (34 decimal digits)
→ use for normal integers	→ use for large integers	→ use for floats where high precision is not reqd	→ use for floats where high precision is required

Note:- MongoDB shell (cmd prompt) is based on JS, its running on JS

```
db.persons.insertOne({age: NumberInt("29")})
```

// In this way we can save memory

// Since shell stores by default numbers as Doubles (64 bit), we can get rid of decimals & reduce size by using NumberInt wrapper.

→ db.companies.insertOne({"valuation": NumberLong("2147483648")})
↳ (64bit int)
→ (Always use quotation marks)

Security & User Authentication

Security Checklist

1. Authentication & Authorization
2. Transport Encryption (Data sent from app to server, shud be encrypted so that no one can spoof)
3. Encryption at Rest (Data in db shud be encrypted)

(Developer Role)

4. Auditing (Server-admin task)
5. Server & Network Config & Setup
6. Backups & S/w updates

(1) Authentication & Authorization

Authentication	Authorization
Identifies valid users of the database	Identifies what these users may actually do in the db.

Role Based Access Control System

→ Mongo DB employs for authentication & authorization

User (Data Analyst) login with username + pwd. Logged in but can't do anything

MongoDB Server

Shop Database	Blog Database	Admin Database
Products coll, Customers coll.	Posts coll, Authors collection.	

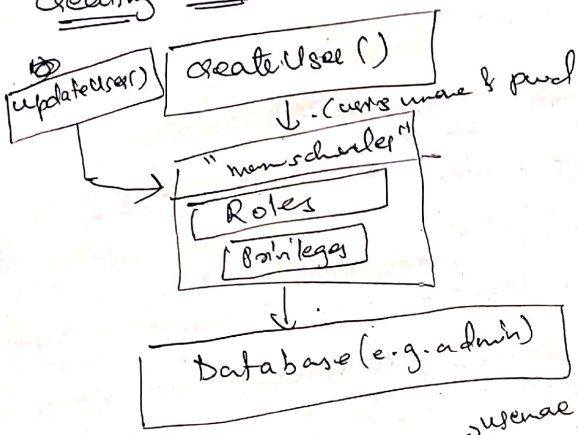
→ users are assigned roles which includes privileges (set of resources & actions)

privileges	
Resources	Action
shop.products	insert()

Why Roles?

1) Different Types of Database users.
(Administrator, Developer, Data Scientist, etc)

Creating & editing users



→ use admin
→ db.createUser ({user: "man", pwd: "man"
roles: ["userAdminAnyDatabase"]})
(built in role which grants user the right to administer any db)

→ db.auth ('man', 'pwd')
username, password.

Built-in Roles

Database user Roles	Database Admin Roles
1) read	1) db Admin
2) readWrite	2) user Admin
	3) dbOwner

All Database Roles

1) readAnyDatabase
2) readWriteAnyDatabase
3) userAdminAnyDatabase
4) dbAdminAnyDatabase

Cluster Admin Roles
(multiple MongoDB servers are working together)

1) clusterManager
2) " Monitor
3) hostManager
4) clusterAdmin

Backup/Restore

1) backup
2) restore

Superuser

1) dbOwner (admin)
2) userAdmin (admin)
3) userAdminAnyDatabase
4) root (most powerful role, which provides user opportunity to do anything)

// See MongoDB documentation

Assigning Roles to users

→ use shop
→ db.createUser({user: 'man1', pwd: 'man1', roles: ["readWrite"]})

→ db.auth('man1', 'man1')
→ db.persons.insertOne({age: 20})

[note: To log out user db :- db.logout()]

Updating & extending roles to other databases

→ db.logout()
→ use as admin db.auth('max', 'max') (As admin db login we can do it)
→ use shop
→ db.updateUser("man1", {roles: ["readWrite", {role: "readWrite", db: "blog"}]})

- "man1" ↓ username
- roles ↓ It replaces earlier roles
- "readWrite" ↓ current role for current db
- role ↓ role for other db
- db ↓ name of db

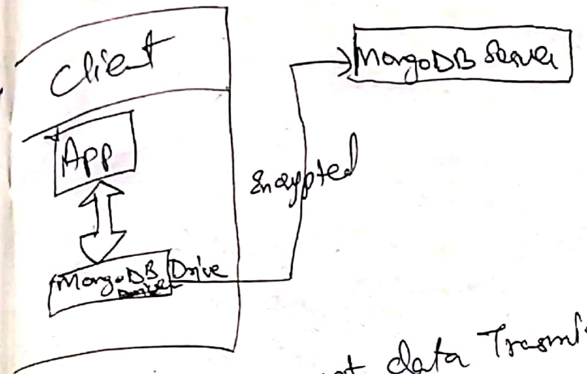
** We can get info. about a particular user:-
db.getUser("man1")

**
→ A user with userAdmin (or) userAdminAnyDatabase role in admin db, can administrate user & roles such as : create users, grant or revoke roles.

Revoke roles

db.revokeRolesFromUser("usename", [roles ...])

(II) Transport Encryption



→ We can encrypt data Transmitted from client to server. (MongoDB uses S.SL (secure socket layer) for encryption.)

~~→ Command fo~~

(III) Encryption at Rest

→ It means data we store in MongoDB server (in a file) will also be encrypted.
→ It is available in Enterprise MongoDB edition.

Sharding (Horizontal Scaling)

→ Requirement of more servers.

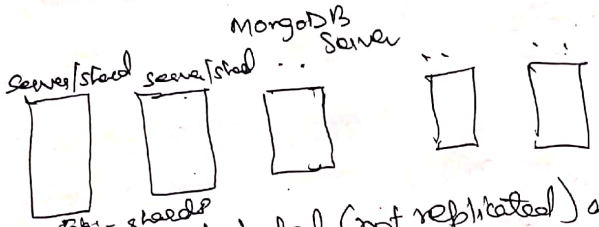


Fig:- shards

→ Data is distributed (not replicated) across shards

→ Queries are run across all shards.

→ we have a router (mongos) which uses shard (keys) to broadcast queries to right shards.

Deploying a MongoDB Server

Local host ⟹ web server / host

(we want a server so that we can access from anywhere not only inside our computer)

→ very complex tasks. We need to manage:-

a) manage shards by Manage Replica sets
c) Secure users/auth setup
d) Protect web server / N/w
e) Update S/w
f) Regular Backups
g) Encryption (Transportation & Rest)

→ MongoDB atlas is a server running on a cloud, which we can configure through a convenient interface, all is available for free, which performs all the tasks above mentioned.

---

using MongoDB atlas

(See later)

Transactions (Either succeed together (or) Fail Together)

User collection ⟹ Posts collection

1 User doc → related → 1 Post Doc

1 should be deleted together → 1 Post doc

→ If a user deletes Account, Post should be deleted too, we can achieve via transaction

-P.T.O

→ for transaction we need sessions (all the requests are grouped together logically)

→ const session = db.getMongo().startSession()

→ session.startTransaction()

→ const userC = session.getDatabase("blog").collection

→ const userP = ... ("blog").collection

→ userC.deleteOne({"_id": 1})
// It doesn't delete from db, but acts as a todo.

→ userP.deleteMany({"_id": 1})

// To commit these changes, we have to do follo

→ session.commitTransaction()

Note :- In order to clearly close the session, use :-

session.abortTransaction()

From Shell to Drivers

Shell	Drivers
1) configure database	1) CRUD operations
2) create collections	2) Aggregation Pipelines
3) create indexes	

Project

⇒ In MongoDB Atlas, create a user under Security → Database Access for read & write roles - (user :- man 1, pwd :- man 1

⇒ Under Security → Network Access → IP white list, add our own IP Address (The place from where we can access cluster)

⇒ After installation of node.js, in cmd prompt :-

→ npm install

→ npm start (start the development server)

(we will do this after React & Node course)