Mongo DB - week 2

Creating Documents :

→ Insert one ()

→ Insert Many () : Has option to do bulk inserts as well as insert ordering to skip failed records.

_id field → mongo creates one id field by default
(Primary Index) if user has not specified one.

Object Id : DATE | MAC ADDR | PID | COUNTER
12 - BYTE HEX STRING

Reading Documents :

* db . movieDetails . find ({rated : "PG-13"}) . Count()
Query Document
(Or) ⇒ 152 records

db . movieDetails . find ({rated : "PG-13", Year : 2009}) . Count()
⇒ 8 records.

⇒ db . movieDetails . find ({"tomato . meter" : 100}) . Pretty()
* ↘ Nested Document find.

Equality matches:

① ON Entire Array:

Ex: db.movieDetails.find({"writers": ["Ethan Coen", "Joel Coen"]}).count()

* Here order of element matters.

② Based on any element:

db.movieDetails.find({"actors": "Jeff Bridges"}).Pretty()

* Will match actors having Jeff Bridges in the movie list.

③ Based on Specific element:

Ex: find movies where Jeff Bridges is the main actor

db.movieDetails.find({"actors.0": "Jeff Bridges"}).Pretty()

* > .0 means first element in actors array.

* find() method returns a cursor

So: var c = db.movieDetails.find

var doc = function() { return c.hasNext() ? c.next() : null; }

c.objsLeftInBatch():

101

doc() → next Element using above function.

Projection: Reduce the Size Of data returned by a Query. (limit the fields returned)

Ex:
```
db.movieDetails.find ({rated : "PG"},
                      {title : }).Pretty()
```
Will Contain Only

* Explicitly exclude Some fields.
```
{title : 1, _id : 0}
```
This will exclude the _id field in results.

Query Selectors:

$eq, $gt, $gte, $Lt, $Lte

Ex: Find all movies having run-time greater then 90.
```
db.movieDetails.find ({runtime : {$gt : 90}}).Pretty()
```

```
* db.movieDetails.find ({runtime : {$gte : 90, $Lte : 120}}).
                                                  Pretty()
db.movieDetails.find ({rated : {$ne : "UNRATED"}}).Count()
db.movieDetails.find ({rated : {$in : ["G", "PG"]}}).Pretty()
```

$exists, $type → If a field is of specified type

Ex:

```
db.movieDetails.find({"tomato.meter" : {$exists : true}}).Count(
```

```
db.movieDetails.find({"_id" : {$type : "String"}}).Count()
```

Logical Operators:

$or, $and, $not, $nor (logical NOR returns all documents that fail to match both clauses)

```
db.movieDetails.find({$or : [{"tomato.meter" : {$gt : 95}},
                {"metacritic" : {$gt : 88}}]}).Pretty()
        (or)
    ({$and : [{"tomato.meter" : {$gt : 95}}, ......
```

* $and mostly used to specify multiple criteria on same field.

Regex Operators:

```
db.movieDetails.find({"awards.text" : {$regex : /^won\s.*/}}
```

Array Operators:

$all → Matches arrays that contain all elements specified in the array.

Ex:

```
db.movieDetails.find({genres: {$all : ["comedy", "Crime", "Drama"]
                    }}).Pretty()
```

```
db.movieDetails.find({countries : {$size : 1 }}).Pretty()
```

$elemMatch

```
db.movieDetails.find({boxOffice : {country : "UK", revenue : {$gt
                                       :15 }}})
```

⇓

This would match the box office element and look for Country UK and revenue greater than 15 in different elements. So use below!

```
db.movieDetails.find({boxOffice : {$elemMatch {
          country : "UK", revenue : {$gt :15} }}})
```

UPDATING DOCUMENTS.

1. Update One
2. Update many
3. Replace one

```
db.movieDetails.UpdateOne ( { title : "The Martian" },
                 { $set : { Poster : "http: // . .......
```

Update field Operators : $rename, $set, $unset

```
db.movieDetails.UpdateOne ( { title : "The martian" },
                 { $inc : { "tomato.reviews" : 3 , "tomato.userReviews" :
                                  25 } } )
```

* For array field updates use $push.

$push along with $each.

$slice → keep only these many elements in the array . For example 5 more recent.

$position → where to push this array element

* Update many → update all documents that match the filter.

```
db.movieDetails.UpdateMany ( { rated : null },
                 { $unset : { rated : "" } } )
```

UPSERTS : → like DB2 merge.

Mongo DB Schema Design.

⇒ Application Driven Schema : 3rd Normal Form.

- * Rich Documents.
- * Pre Join / Embed Data
- * NO Joins in mongo DB (Embed directly)
- * No Constraints.
- * Atomic Operations
- * NO Transaction Support.
- * NO Declared Schema.

* Avoid Constraints & go for embedding Data using Collections.

* Atomic Operations (i.e all or nothing) in mongo DB instead Of Transactions.

One to one Relations

★ Example : Employee ⟷ Resume

When to have it in seperate Collection. (i.e to embed or not)
- → Frequency of Access
- → Size of items (MongoDB doc 16MB)
- → Atomicity of data

One to Many Relations.

Example : City to Persons.
(NY) → (8 million People)
blog Posts : Comments
(1) : (10)

(one Scenario) Posts vs People

```
Posts                    People
{ name                   { name : "Sami",
  Comments : [             City : "NYC",
    — —                  }
    — —                        and
    — —                  City.
  ]                      { _id : "NYC"
}                          —
                           —
                         }
```

⟹ one to few Scenario.

Many to Many Relations :

Ex: Books : Authors
Students : teachers

⟹ Few : Few.

Authors
_id : 27
author_name : "mitchel Stark"
books : [12, 7, 8]

Books.
_id : 12
title : "Gone with the wind"

Multi key Indexes :

Students
{ _id : 0
name : 'Andrew',
teachers : [1, 7, 10, 23] }

teachers.
{ _id : 10,
name : 'Tony Stark' }

db.Students.find({'teachers' : {$ai : [0,1]}})

and

db.Students.find({'teachers' : {$all : [0,1]}}).explain() ← ✱ What it did while the query got executed.

✱ cursor : Btree Cursor
isMultikey : true.

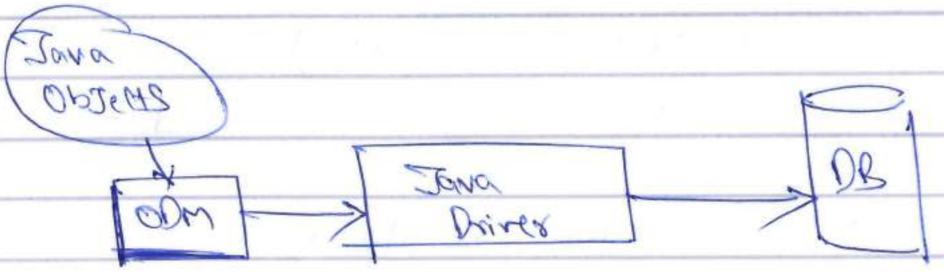
Benefits Of Embedding :

→ Improved Read Performance

→ One Round trip to DB

* 3NF will avoid modification anamolies.

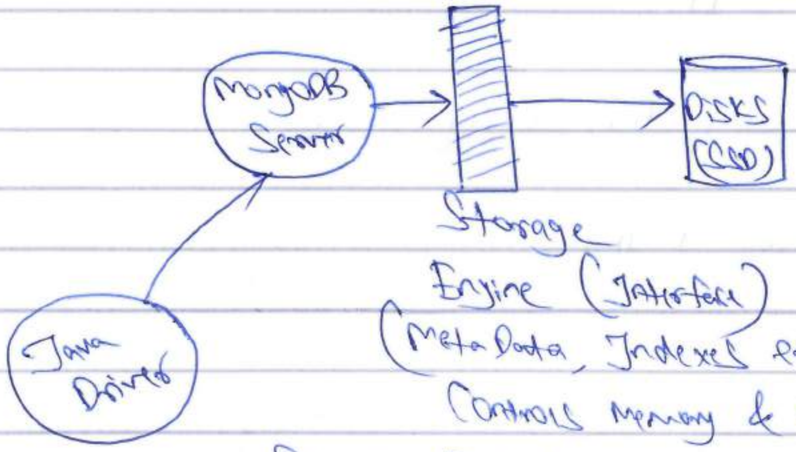
ODM – Object Document Mapper.



* Morphia → Java ODM for MongoDB.

Mongo DB (week 4)

→ Pluggable Storage Engines.



Storage Engine (Interface)
(Meta Data, Indexes etc)
Controls Memory & when to Persist to Disk.

Storage Engines:
- * MMAP → Collection level locking.
- * Wired Tiger → Not Default.
  - * Offers Document level concurrency.
  - * Compression Of documents & indexes
  - * No Inplace Update

Index

* Ordered list of things having Pointer to Original record on the Disk.
* BTree implementation.

Writes : Slower } Indexes
Reads : Faster } Disadvantage.

Ascending

db.Students.CreateIndex ({Student_id : 1}); −1 = Descending.

★ db.Students.Explain().find ({Student_id : 5});
Query Plan on the Collection followed by the Command. And winning Plan Can be Checked.

* Explain(true) ⟹ To Check how many documents were Examined.

db.Students.getIndexes(); ⟹ To Check the index on a Collection
↳ By Default index will be on _id
db.Students.dropIndex ({Student_id :1});

Multi key Indexes

Eg: { name : 'Andrew',
tags : ['Photography', 'hiking', 'golf'],
Color : 'red',
location : ['NY', 'CA']}

Index on tags or (tags, Color)

But (tags, location) cant be index because both tags & location is an array document.

- Notation & multi key.

* db. Students. CreateIndex ({'Scores. Score' : 1});

Unique Indexes :

db. Stuff. CreateIndex ({thing : 1}, {"unique" : true});

* Sparse Indexes → when index key is missing from some of the documents.

For example:

{a:1, b:2, c:5}	Index on
{a:10, b:5, c:10}	a → possible
{a:13, b:17}	b → possible
{a:7, b:23}	c → Not possible since c value not in some docs.

* This is when sparse index helps which wont include document 3 and 4 in index since it does not have c value.

⇒ db. employees. createIndex ({cell : 1}, {unique: true,
sparse : true});

Sparse Index ⟶ uses collection scan, so dont
use for sorting.

Background Indexes:

db. Students. createIndex ({'scores. score':1}, {background: true}

⇒ Slow
⇒ Dont block readers | writers.

MongoDB explain:

db. foo. explain (). help ()

☆ Var exp = db. example . explain()

Buffered: exp. help ();
exp. find ({a:17, b:55}). Sort ({b : -1});

Explain:

→ Query Planner (Default)
→ Execution Stats
→ allPlans Execution

```
var exp = db.example.explain("executionStats");
    exp.find({a:17, b:55});
```

* Winning Plan + ReJected Plans + Detailed info regarding the several stages of Query execution.

```
var exp = db.example.explain("allPlansExecution");
```

* Execution stats for all Plans.
  ⇒ allPlansExecution array of all Plans.

Covered Indexes:

```
exp.find({i:45, j:23}, {_id:0, i:1, j:1, k:1});
```

↳ Don't include _id in the display.

```
exp.find({i:45, j:23}, {_id:0});
```

↳ more documents examined as mongoDB don't know how many more variables are there like i, j or k e.t.c.

Choosing an Index:

Example Indexes : ① b, c ⇒ Potential Candidates so three query plan
② c, b and check which is fastest.
d, e ⇒ winning criteria
e, f ① Returned all results
③ a, b, c ② Threshold values.
③ Cache winning query.

Threshold writes — Rebuild — mongodb restart

db.collection.stats()
↳ will give total Index Size()
(or) db.collection.totalIndexSize()

✯ So make sure your Indexes fits in the memory.

Index Cardinality.

Regular	Sparse	Multi key
1:1	≤ documents	tags : [—, —, —] > documents.

Geo Spatial Indexes.

⇒ find Things based on location.

i.e Search based on location.

```
* db.Collection.ensureIndex({"location":'2d', type:1});
                                                  ↘ Ascending
⇒ 2 Dimensional Index.
```

```
db.Collection.find({location:{$near:[50,50]}})
        (or)
db.Collection.find({location:{$near:[50,50]}}).limit(3);
```

GeoSpacial Spherical.

* 2d Sphere

* GeoJSON for latitode, longitude Coordinates.

```
db.Collection.ensureIndex({'location':'2dSphere'})
                                         ⇓
                                  For GeoJSON Document
                                       Index.
```

Example Query:

```
db.Places.find({
      location:{
          $near: {
              $geometry: {
                  type: "Point",
                  Coordinates: [-122.16, 37.42]},
                  $maxDistance: 2000 }
          }
   }) }
```

Full Text Search.

db.Collection.ensureIndex({'words':'text'}) → text Index. ✱✱

Example : db.Sentences.find({$text:{$Search:'dog'}})

text Score ⇒ Document with more word Sucessfully matched.

Ex:

```
db.Sentences.find({$text:{$Search:'dog tree Obsidian'}},
    {Score:{$meta:'textScore'}}).Sort({Score:{$meta:
                                   'textScore'}})
```

Designing Using Indexes.

Goal : Efficient Read/write Operations.

logging & Profiling :

⇒ logs Slow Queries by default for Query running more than 100 ms.

Profiler : Levels

0	1	2
Off	log Slow Queries	All my Queries.

* db. System. Profile. find ({millis : {$gt : 1}}). Sort ({ts : 1}). Pretty ();
  ↳ Query which is greater than 1ms and Sorted by time Stamp ~~Descending~~. Ascending.

→ db. get ProfilingLevel () ⇒ To get level of logging.
→ db. Set Profiling ~~Status~~ level (1, 4)

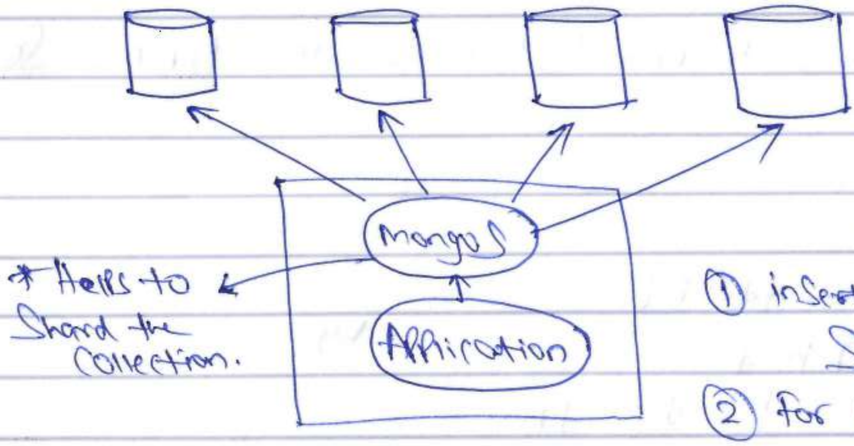
Mongo top.

* Mongo DB Profiler. with read & write time information.

Mongo Stat → What is going on during insert, update, delete queries.

SHARDING:

⇒ Splitting up large collection among multiple Servers.



* Based on the Query Send request to the right Server instence.

① insert Shard have a Shard key.

② for update/remove/find if Shard key is not there then it Sends requests to all Servers (i.e broad casting)

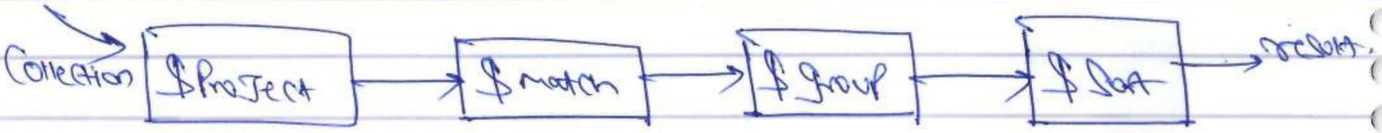
Week 5 — Aggregation.

⇒ Similar to Count, groupby aggregation in RDBMS SQL.

MongoDB example:

```
db.Products.aggregate ([
        {$group :                  key
              {_id : "$manufacturer"
                num_Products : {$sum:1}
        }     }
      ])
```

Aggregation Pipeline:



$Project — reShape a document — 1:1 (one document will leave)

$match — filter Stage — n:1

$group — aggregate — n:1 ($sum or $count)

$Sort — Sort — 1:1

$Skip — Skips — n:1

$limit — limits — n:1

$unwind — Flatten the data

Ex: Initial document tags: ["red", "blue", "green"]

⇓

tags: red, tags: blue, tags: green

Compound Grouping:

```
_id: { "manufactorer" : "$manufactorer"
       "Category" : "$Category" },
  num_products : { $sum : 1 }
```

★ _id field can be a document also. and not always a Scalar value.

Aggregation Expressions Overview.

$sum, $avg, $min, $max, $push] Push to
$addtoSet] Array

$first] Need to first Sort the document Otherwise
$last] it will be Arbitrary.

$addToSet → Add to collection if its not there.

```
db.Products.aggregate([
        {$group:
            {_id: { "maker" : "$manufacturer"},
                Categories: {$addToSet : "$category"}
        }
    }])
```

☆☆" maker wise find out all the Categories.

<u>Double Grouping:</u>

```
    db.fun.find()
       {"_id":0, "a":0, "b":0, "c":21}
       {"_id": 1, "a":0, "b":0, "c":54}
         . . . . ..

db.fun.aggregate([{$group: {_id: {a: "$a", b: "$b"},
                                C: {$max: "$c"}}},
        {$group : {_id: "$_id.a", C: {$min, "$c"}}}])
```

$Project :

- remove keys
- Add keys
- use some simple functions
    - $toUpper
    - $tolower
    - $add
    - $multiply.

Example :

```
db.Products.aggregate ([
    { $Project :
        { _id : 0  → exclude _id field
          'maker' : { $toLower: "$manufacturer" },
          'details' : { 'category' : "$category",
                        'Price' : { "$multiply : ("$Price :
                      },
          'item' : '$name' }}])
```

$Sort :

⇒ disk & memory based.
→ Default is in-memory based. (100 MB)

⇒ before or after the grouping stage.

Double unwind :

* → when there are two arrays in a document.

→ Can use $push to reverse the impact of two $unwinds.

*

RDBMS	Mongo DB
WHERE	— $match
HAVING	— $match
SELECT	— $project
ORDER BY	— $sort
SUM()	— $sum
JOIN	— mostly using $unwind.

limitations in Aggregation.

① 100 MB limit for Pipeline — use allowDiskUse for this.

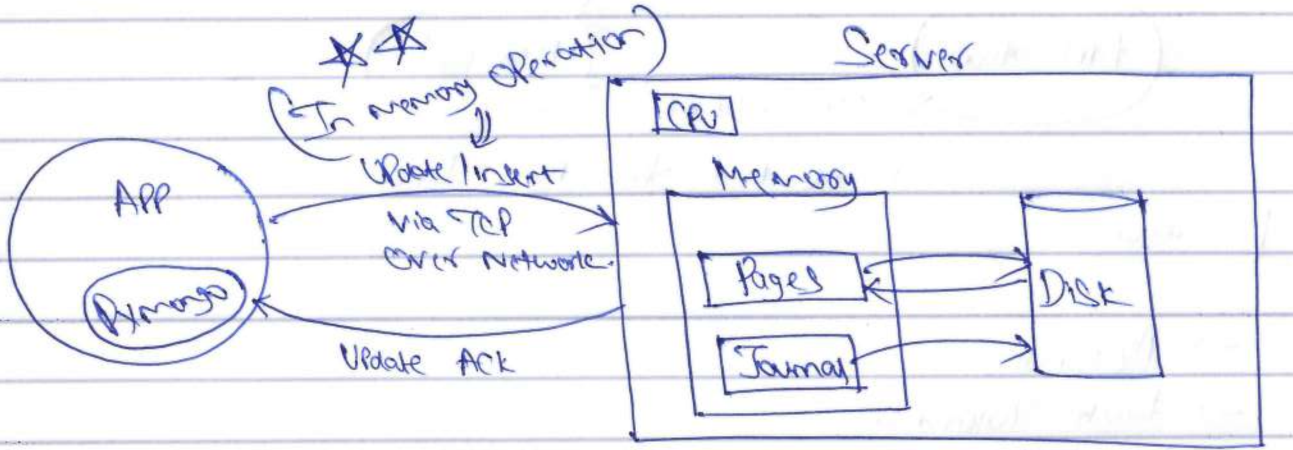
② 16 MB limit for document size — use cursor = { ... }

③ Sharded → group by, sort will cause it to work on the Primary shard and not the replicas hence limiting scalibility when using aggregation

⇒ use Hadoop — map / Reduce using mongo hadoop connector.

Week 6

Write Concern.



* Updates / inserts first go and sits in the Pages / Journal area for a While before it goes to disk.

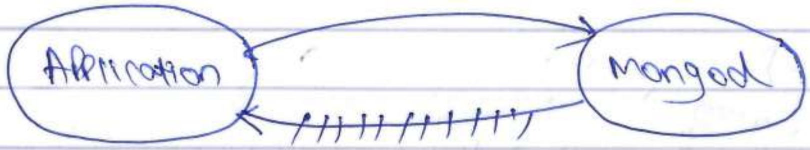
* Acknowledge ← $\boxed{\begin{array}{l}\text{Write} = 1 \\ \text{Journal} = \text{false}\end{array}}$ ⟹ Write Conform.

* If there is a server crash and if the Journal Contents were not written to disk, then the data is lost.

w	j	
1	false	→ fast, but small window of vulnerability.
1	True	→ slow and can be mentioned @ driver level.

Network Errors.

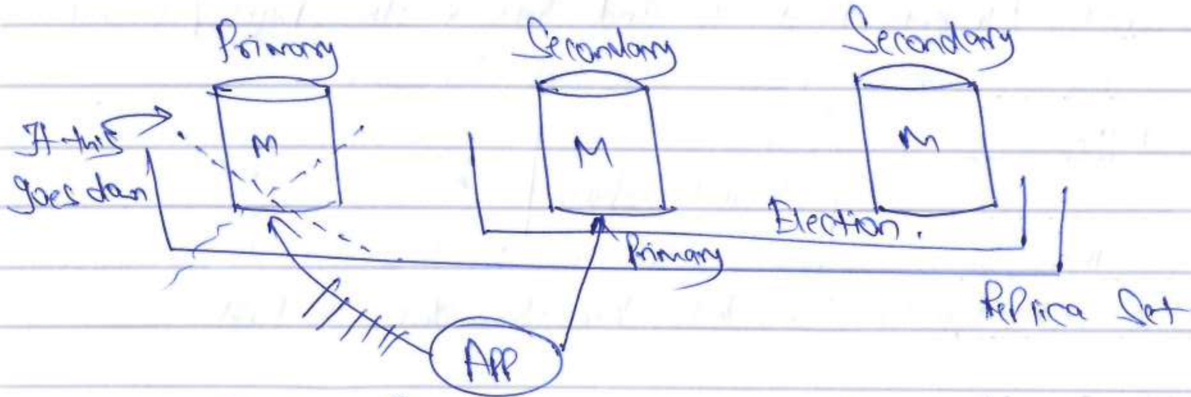
When w=1 & j=true



If no ACK then N/w Errors.

Replication

→ Availibility
→ Fault Tolerance.



* If the Primary which went down comes back up, then this will Join as the Secondary node
* Minimum number of nodes is 3. i.e to have the election Process.

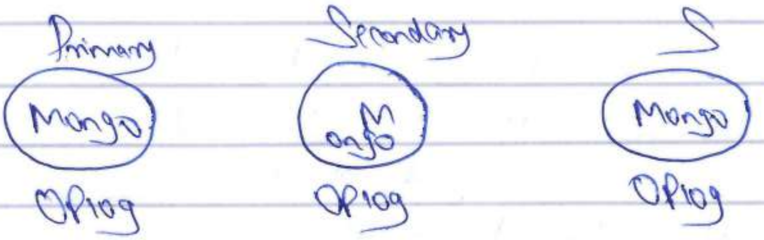
* Replication is Asynchronous.

Types of Replica Set Nodes.

* Regular
* Arbiter – There for voting Purpose
* Delayed | Regular ← Disaster Recovery node – Can't be in voting.
  ↳ Cannot be Primary (Priority = 0)
* Hidden – Used for Analytics. Priority = 0.

⇒ During Primary Down, no writes allowed.

Replication Internals.



* Write on Primary will give information in Oplog and the Secondary will query this Oplog and apply the Same Operation on itself.

Oplog : db.oplog.rs.find().pretty();

* Secondary will have the Optime and OptimeDate which lets us know the latest Update Date / Time.

rs.getStatus(); ⇒ Check Syncing To also.

** If a node comes back up as a Secondary after a Period of being Offline and the Oplog has looped / gone ahead on the Primary, then

⇒ The entire data set will be copied from the Primary.

Re-visiting Write Concerns.

W=1 – wait for Primary to Ack the write
W=2 – wait for Primary & one Other Secondary
W=3 – —— " —— & all Other Secondary

Wtime out – How long to wait?

→ Set on : (1) Connection
(2) on a Collection
(3) Replica Set → Safest to do.

W= MaJority → wait for MaJority of nodes to Ack.

** Journal waits for Primary nodes only.

Read Preference

→ Default read Preference is Primary. To read from Secondary, Use rs.SlaveOk();

→ Primary Preffered – Primary is Preffered if not there then read from Secondary.

Eventually consistent Read.

→ ⇓ Secondary – Read from Secondary

→ Secondary Preffered – Secondary is Preffered, if not then Primary

→ Nearest – Based on min Ping time.

Implication Of Replications:

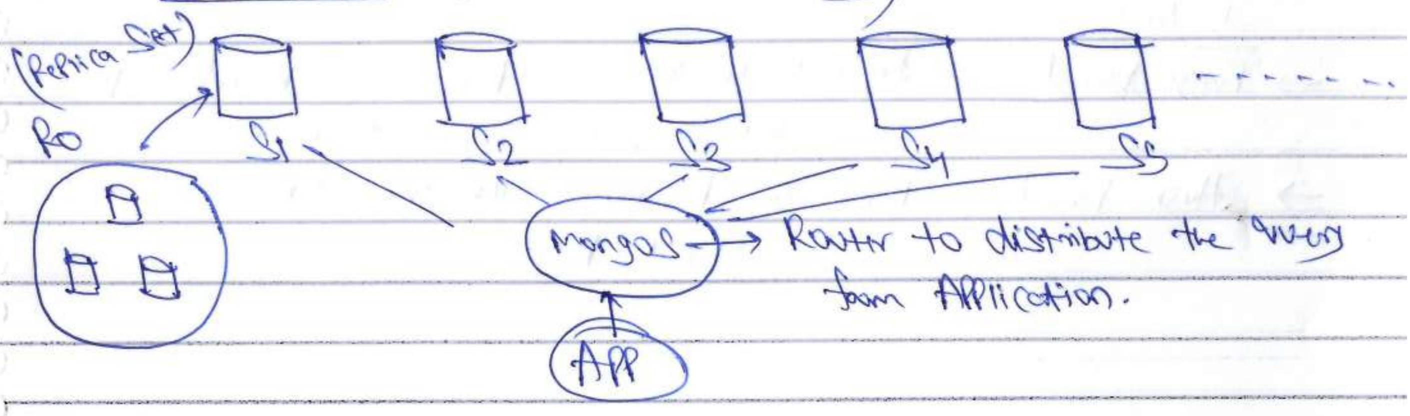
* Seed lists – Node should be aware Of Other nodes

* Write Concern – w & j Parameter also wtimeout

* Read Preferences – Since multiple Options are there

* Errors can happen 😵 – N/w errors, Exceptions, Syntax issues etc

SHARDING (Horizontal Scalability)



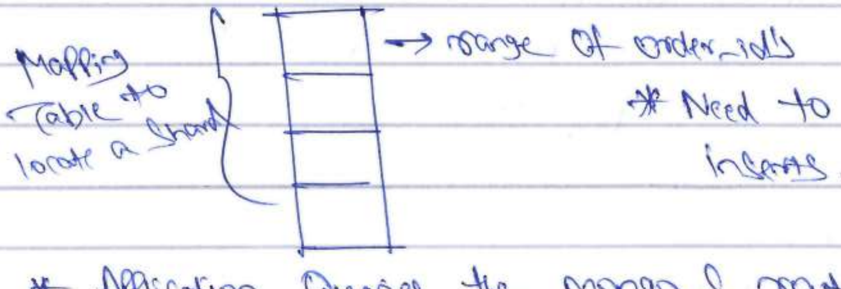
* Break up the Collection into multiple logical parts and have a Shard key.

→ Range based Sharding

→ Shard key

Ex: Orders Table

* Breaked into chunks.



* Need to include Shard key on inserts.

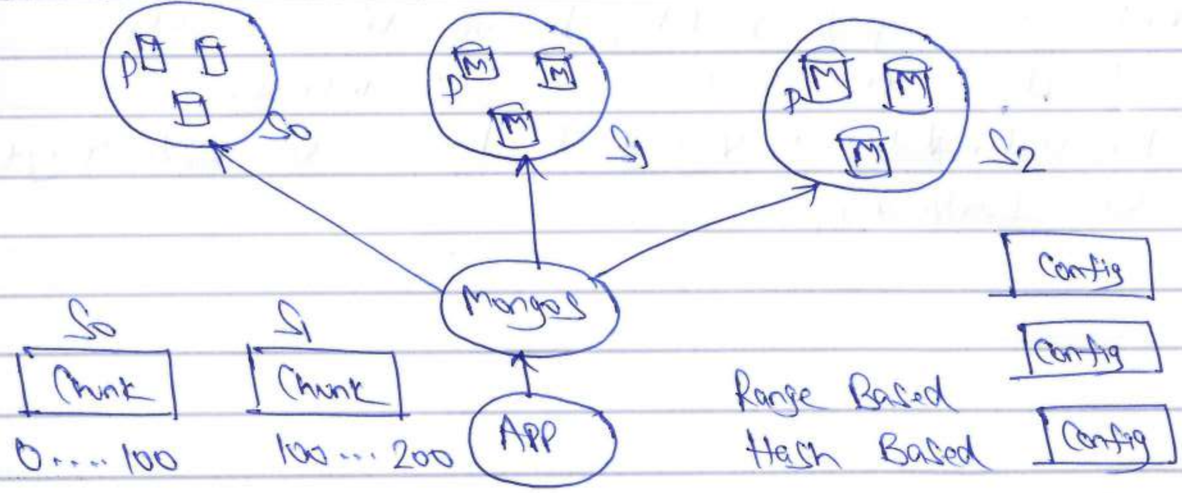
* Application Queries the mongo S router which will find out a particular Shard where the request could be fulfilled.

* Multiple mongoS can be present.
* Each Shard is a replica Set again.

Sharding.

→ Range based – Ex: 0 to 100 in particular Shard S1
101 to 200 in S2 etc.

→ Hash based – Runs a hashing algorithm on the Shard key.



* Shard fail over → mongos will try to re connect.

Choosing a Shard key.

* Sufficient Cardinality – i.e number of Possible values in a Colum.

* Avoid Hot spotting in writes. i.e Problem of writes always hitting the Same Shard because of the Shard key. So choose a Shard key which is not Just increasing in nature but have sufficient Cardinality

Ex: Orders

{ Order_id ; ——,
Order_date ; ——,
Vendor ; —— }

① vendor is a good Shard key because Of Cardinality.
② Order_id is not great as its Just increasing.
③ (Vendor, Order_date) is very good because of wide range Of Cardinality.