

```

1  -----
2  01-important-note
3  -----
4  // Definitions
5  /**
6   * # store data as documents like JavaScript Object
7   * # noSQL database
8   * # store like JSON
9   * # BSON (b-> binary) Data Structure into server
10 */
11
12
13 No Schema!
14 User Collection
15 id: 1 "name": "Raju" "age": 22 ...
16
17 /**
18 Relations
19 No / Few Relations
20 #Relational Data needs to be merger manually-- > kind of
21
22 #flow
23 1. Database
24 2. Collections (like tables)
25 3. Documents (like JSON Object values)-these are schemaless
26
27 #MongoDB Ecosystem
28
29 MongoDB Database
30 1. Self - Managed / Enterprise      2. Atlas (Cloud)      3. Mobile
31   |
32   |
33   CloudManager / OpsManager
34 // other options
35 1. Compass
36 2. BI Connectors
37 3. MongoDB Charts
38 */
39
40 Stich --> Basically a serverless backend solution
41 This gives :
42 1. serverless query api -> a tool sets or tools to directly database query from
43    inside client apps.
44 2. Server Functions (in the cloud-> related to js)-> like google cloud function
45    or AWS Lambda.
46 3. Database Triggers -> that allows us to events in a database, like document was
47    inserted and then
48    execute a function in response to that and that function could then maybe
49    send an e-mail.
50 4. Real-Time Sync -> basically is built to synchronize a database in a cloud with
51    that mobile offline supporting database.
52
53 Working with MongoDB
54 Backend Server -> Drivers (Node.js, Java, Python) or MongoDB
55 Shell->(queries)<->(communicate) MongoDB Server->Storage Engine (File/Data Access)
56
57           | Read + Write Data to Files (slow) ->when Database
58 Storage Engine |
59           | Read + Write Data to Files (fast) ->when Memory
60
61 # documents created implicitly
62
63 JSON data converts into BSON data
64
65 {
66   "name" : "MAX",

```

```

62     "age" : 29
63 }
64
65 it converts into BSON
66
67 BSON 1. Binary data 2. Extends JSON Types(e.g more detailed Number Types) 3.
Efficient Storage
68
69
70
71 -----
72 02-database-operations-basic
73 -----
74
75 # show all database
76 show dbs
77
78 # create a new database
79 > use shop
80 switched to db shop
81
82 > db.products.insertOne({name: "A Book", price: 12.99})
83
84 > db.products.find()
85 { "_id" : ObjectId("5f12a043feace8e519f293ed"), "name" : "A Book", "price" : 12.99 }
86
87 > db.products.find().pretty()
88 {
89     "_id" : ObjectId("5f12a043feace8e519f293ed"),
90     "name" : "A Book",
91     "price" : 12.99
92 }
93
94
95 > db.products.insertOne({name: "A T-shirt", price: 29.99, description: "This is
high quality T-shirt"})
96 {
97     "acknowledged" : true,
98     "insertedId" : ObjectId("5f12e3b1fa9f127dd8b9c7b4")
99 }
100 > db.products.find().pretty()
101 {
102     "_id" : ObjectId("5f12a043feace8e519f293ed"),
103     "name" : "A Book",
104     "price" : 12.99
105 }
106 {
107     "_id" : ObjectId("5f12e3b1fa9f127dd8b9c7b4"),
108     "name" : "A T-shirt",
109     "price" : 29.99,
110     "description" : "This is high quality T-shirt"
111 }
112
113 > db.products.insertOne({name: "A Computer", price:34829.99, description: "This is
high quality computer", details:{cpu: "Intel i7 8770",memory: 32}})
114 {
115     "acknowledged" : true,
116     "insertedId" : ObjectId("5f12e5edfa9f127dd8b9c7b5")
117 }
118 > db.products.find().pretty()
119 {
120     "_id" : ObjectId("5f12a043feace8e519f293ed"),
121     "name" : "A Book",
122     "price" : 12.99
123 }
124 {
125     "_id" : ObjectId("5f12e3b1fa9f127dd8b9c7b4"),

```

```

126     "name" : "A T-shirt",
127     "price" : 29.99,
128     "description" : "This is high quality T-shirt"
129 }
130 {
131     "_id" : ObjectId("5f12e5edfa9f127dd8b9c7b5"),
132     "name" : "A Computer",
133     "price" : 34829.99,
134     "description" : "This is high quality computer",
135     "details" : {
136         "cpu" : "Intel i7 8770",
137         "memory" : 32
138     }
139 }
140
141 // to create a different port
142 sudo mongod --port 27018
143 // to start port
144 mongo --port 27018
145
146
147 > db.flightData.insertOne( {      "departureAirport": "MUC",      "arrivalAirport":
"SFO",      "aircraft": "Airbus A380",      "distance": 12000,
"intercontinental": true  })
148 {
149     "acknowledged" : true,
150     "insertedId" : ObjectId("5f130951d022deabe244f26c")
151 }
152 > db.flightData.find().pretty()
153 {
154     "_id" : ObjectId("5f130951d022deabe244f26c"),
155     "departureAirport" : "MUC",
156     "arrivalAirport" : "SFO",
157     "aircraft" : "Airbus A380",
158     "distance" : 12000,
159     "intercontinental" : true
160 }
161
162
163 -----CRUD Operations-----
164
165 // Create
166 insertOne(data, options)
167 insertMany(data, options)
168
169 > db.flightData.insertMany([
170     {
171         "departureAirport": "MUC",
172         "arrivalAirport": "SFO",
173         "aircraft": "Airbus A380",
174         "distance": 12000,
175         "intercontinental": true
176     },
177     {
178         "departureAirport": "LHR",
179         "arrivalAirport": "TXL",
180         "aircraft": "Airbus A320",
181         "distance": 950,
182         "intercontinental": false
183     }
184 ])
185 {
186     "acknowledged" : true,
187     "insertedIds" : [
188         ObjectId("5f132aebd022deabe244f26d"),
189         ObjectId("5f132aebd022deabe244f26e")
190     ]
191 }
192 > db.flightData.find().pretty()

```

```

193 {
194   "_id" : ObjectId("5f132aebd022deabe244f26d"),
195   "departureAirport" : "MUC",
196   "arrivalAirport" : "SFO",
197   "aircraft" : "Airbus A380",
198   "distance" : 12000,
199   "intercontinental" : true
200 }
201 {
202   "_id" : ObjectId("5f132aebd022deabe244f26e"),
203   "departureAirport" : "LHR",
204   "arrivalAirport" : "TXL",
205   "aircraft" : "Airbus A320",
206   "distance" : 950,
207   "intercontinental" : false
208 }
209 >
210
211 // Read
212 find(filter, options)
213 findOne(filter, options)
214
215 find gives us a cursor object not an array
216
217 > db.flightData.find({intercontinental: true}).pretty()
218 {
219   "_id" : ObjectId("5f132aebd022deabe244f26d"),
220   "departureAirport" : "MUC",
221   "arrivalAirport" : "SFO",
222   "aircraft" : "Airbus A380",
223   "distance" : 12000,
224   "intercontinental" : true
225 }
226
227 > db.flightData.find({distance: {$gt: 10000}}).pretty()
228 {
229   "_id" : ObjectId("5f132aebd022deabe244f26d"),
230   "departureAirport" : "MUC",
231   "arrivalAirport" : "SFO",
232   "aircraft" : "Airbus A380",
233   "distance" : 12000,
234   "intercontinental" : true
235 }
236
237 > db.flightData.findOne({distance: {$gt: 900}})
238 {
239   "_id" : ObjectId("5f132aebd022deabe244f26d"),
240   "departureAirport" : "MUC",
241   "arrivalAirport" : "SFO",
242   "aircraft" : "Airbus A380",
243   "distance" : 12000,
244   "intercontinental" : true
245 }
246
247 > db.passengers.find().toArray()
248 [
249   {
250     "_id" : ObjectId("5f1339f7d022deabe244f26f"),
251     "name" : "Max Schwarzmüller",
252     "age" : 29
253   },
254   {
255     "_id" : ObjectId("5f1339f7d022deabe244f270"),
256     "name" : "Manu Lorenz",
257     "age" : 30
258   },
259   {
260     "_id" : ObjectId("5f1339f7d022deabe244f271"),

```

```
262         "name" : "Chris Hayton",
263         "age" : 35
264     },
265     {
266         "_id" : ObjectId("5f1339f7d022deabe244f272") ,
267         "name" : "Sandeep Kumar",
268         "age" : 28
269     },
270     {
271         "_id" : ObjectId("5f1339f7d022deabe244f273") ,
272         "name" : "Maria Jones",
273         "age" : 30
274     },
275     {
276         "_id" : ObjectId("5f1339f7d022deabe244f274") ,
277         "name" : "Alexandra Maier",
278         "age" : 27
279     },
280     {
281         "_id" : ObjectId("5f1339f7d022deabe244f275") ,
282         "name" : "Dr. Phil Evans",
283         "age" : 47
284     },
285     {
286         "_id" : ObjectId("5f1339f7d022deabe244f276") ,
287         "name" : "Sandra Brugge",
288         "age" : 33
289     },
290     {
291         "_id" : ObjectId("5f1339f7d022deabe244f277") ,
292         "name" : "Elisabeth Mayr",
293         "age" : 29
294     },
295     {
296         "_id" : ObjectId("5f1339f7d022deabe244f278") ,
297         "name" : "Frank Cube",
298         "age" : 41
299     },
300     {
301         "_id" : ObjectId("5f1339f7d022deabe244f279") ,
302         "name" : "Karandeep Alun",
303         "age" : 48
304     },
305     {
306         "_id" : ObjectId("5f1339f7d022deabe244f27a") ,
307         "name" : "Michaela Drayer",
308         "age" : 39
309     },
310     {
311         "_id" : ObjectId("5f1339f7d022deabe244f27b") ,
312         "name" : "Bernd Hoftstadt",
313         "age" : 22
314     },
315     {
316         "_id" : ObjectId("5f1339f7d022deabe244f27c") ,
317         "name" : "Scott Tolib",
318         "age" : 44
319     },
320     {
321         "_id" : ObjectId("5f1339f7d022deabe244f27d") ,
322         "name" : "Freddy Melder",
323         "age" : 41
324     },
325     {
326         "_id" : ObjectId("5f1339f7d022deabe244f27e") ,
327         "name" : "Alexis Bohed",
328         "age" : 35
329     },
330     {
```

```

331         "_id" : ObjectId("5f1339f7d022deabe244f27f"),
332         "name" : "Melanie Palace",
333         "age" : 27
334     },
335     {
336         "_id" : ObjectId("5f1339f7d022deabe244f280"),
337         "name" : "Armin Glutch",
338         "age" : 35
339     },
340     {
341         "_id" : ObjectId("5f1339f7d022deabe244f281"),
342         "name" : "Klaus Arber",
343         "age" : 53
344     },
345     {
346         "_id" : ObjectId("5f1339f7d022deabe244f282"),
347         "name" : "Albert Twostone",
348         "age" : 68
349     },
350     {
351         "_id" : ObjectId("5f1339f7d022deabe244f283"),
352         "name" : "Gordon Black",
353         "age" : 38
354     }
355 ]
356
357
358 > db.passengers.find().forEach((passengerData) => {printjson(passengerData)})
359 {
360     "_id" : ObjectId("5f1339f7d022deabe244f26f"),
361     "name" : "Max SchwarzmueLLer",
362     "age" : 29
363 }
364 {
365     "_id" : ObjectId("5f1339f7d022deabe244f270"),
366     "name" : "Manu Lorenz",
367     "age" : 30
368 }
369 {
370     "_id" : ObjectId("5f1339f7d022deabe244f271"),
371     "name" : "Chris Hayton",
372     "age" : 35
373 }
374 {
375     "_id" : ObjectId("5f1339f7d022deabe244f272"),
376     "name" : "Sandeep Kumar",
377     "age" : 28
378 }
379 {
380     "_id" : ObjectId("5f1339f7d022deabe244f273"),
381     "name" : "Maria Jones",
382     "age" : 30
383 }
384 {
385     "_id" : ObjectId("5f1339f7d022deabe244f274"),
386     "name" : "Alexandra Maier",
387     "age" : 27
388 }
389 {
390     "_id" : ObjectId("5f1339f7d022deabe244f275"),
391     "name" : "Dr. Phil Evans",
392     "age" : 47
393 }
394 {
395     "_id" : ObjectId("5f1339f7d022deabe244f276"),
396     "name" : "Sandra Brugge",
397     "age" : 33
398 }
399 {

```

```

400         "_id" : ObjectId("5f1339f7d022deabe244f277"),
401         "name" : "Elisabeth Mayr",
402         "age" : 29
403     }
404     {
405         "_id" : ObjectId("5f1339f7d022deabe244f278"),
406         "name" : "Frank Cube",
407         "age" : 41
408     }
409     {
410         "_id" : ObjectId("5f1339f7d022deabe244f279"),
411         "name" : "Karandeep Alun",
412         "age" : 48
413     }
414     {
415         "_id" : ObjectId("5f1339f7d022deabe244f27a"),
416         "name" : "Michaela Drayer",
417         "age" : 39
418     }
419     {
420         "_id" : ObjectId("5f1339f7d022deabe244f27b"),
421         "name" : "Bernd Hoftstadt",
422         "age" : 22
423     }
424     {
425         "_id" : ObjectId("5f1339f7d022deabe244f27c"),
426         "name" : "Scott Tolib",
427         "age" : 44
428     }
429     {
430         "_id" : ObjectId("5f1339f7d022deabe244f27d"),
431         "name" : "Freddy Melfer",
432         "age" : 41
433     }
434     {
435         "_id" : ObjectId("5f1339f7d022deabe244f27e"),
436         "name" : "Alexis Bohed",
437         "age" : 35
438     }
439     {
440         "_id" : ObjectId("5f1339f7d022deabe244f27f"),
441         "name" : "Melanie Palace",
442         "age" : 27
443     }
444     {
445         "_id" : ObjectId("5f1339f7d022deabe244f280"),
446         "name" : "Armin Glutch",
447         "age" : 35
448     }
449     {
450         "_id" : ObjectId("5f1339f7d022deabe244f281"),
451         "name" : "Klaus Arber",
452         "age" : 53
453     }
454     {
455         "_id" : ObjectId("5f1339f7d022deabe244f282"),
456         "name" : "Albert Twostone",
457         "age" : 68
458     }
459     {
460         "_id" : ObjectId("5f1339f7d022deabe244f283"),
461         "name" : "Gordon Black",
462         "age" : 38
463     }
464
465
466 // Update
467 updateOne(filter, data, options)
468 updateMany(filter, data, options)

```

```

469     replaceOne(filter, data, options)
470
471 > db.flightData.updateOne({distance: 1200},{ $set:{marker: "delete"}})
472 > db.flightData.updateMany({},{$set:{marker: "toDelete"}})
473
474 > db.flightData.updateOne({_id :
ObjectId("5f132aebd022deabe244f26d")},{ $set:{delayed: true}})
475 { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
476 > db.flightData.find().pretty()
477 {
478   "_id" : ObjectId("5f132aebd022deabe244f26d"),
479   "departureAirport" : "MUC",
480   "arrivalAirport" : "SFO",
481   "aircraft" : "Airbus A380",
482   "distance" : 12000,
483   "intercontinental" : true,
484   "delayed" : true
485 }
486 {
487   "_id" : ObjectId("5f132aebd022deabe244f26e"),
488   "departureAirport" : "LHR",
489   "arrivalAirport" : "TXL",
490   "aircraft" : "Airbus A320",
491   "distance" : 950,
492   "intercontinental" : false
493 }
494
495 > db.flightData.update({_id : ObjectId("5f132aebd022deabe244f26d")},{ $set:{delayed:
false}})
496 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
497 > db.flightData.find().pretty()
498 {
499   "_id" : ObjectId("5f132aebd022deabe244f26d"),
500   "departureAirport" : "MUC",
501   "arrivalAirport" : "SFO",
502   "aircraft" : "Airbus A380",
503   "distance" : 12000,
504   "intercontinental" : true,
505   "delayed" : false
506 }
507 {
508   "_id" : ObjectId("5f132aebd022deabe244f26e"),
509   "departureAirport" : "LHR",
510   "arrivalAirport" : "TXL",
511   "aircraft" : "Airbus A320",
512   "distance" : 950,
513   "intercontinental" : false
514 }
515
516 > db.flightData.update({_id : ObjectId("5f132aebd022deabe244f26d")},{delayed: false})
517 WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
518 > db.flightData.find().pretty()
519 { "_id" : ObjectId("5f132aebd022deabe244f26d"), "delayed" : false }
520 {
521   "_id" : ObjectId("5f132aebd022deabe244f26e"),
522   "departureAirport" : "LHR",
523   "arrivalAirport" : "TXL",
524   "aircraft" : "Airbus A320",
525   "distance" : 950,
526   "intercontinental" : false
527 }
528 // same can be done by replaceOne
529 > db.flightData.replaceOne({_id : ObjectId("5f132aebd022deabe244f26d")},{
530   ...   "departureAirport": "MUC",
531   ...   "arrivalAirport": "SFO",
532   ...   "aircraft": "Airbus A380",
533   ...   "distance": 12000,
534   ...   "intercontinental": true
535   ... })

```



```

536 { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
537 > db.flightData.find().pretty()
538 {
539   "_id" : ObjectId("5f132aebd022deabe244f26d"),
540   "departureAirport" : "MUC",
541   "arrivalAirport" : "SFO",
542   "aircraft" : "Airbus A380",
543   "distance" : 12000,
544   "intercontinental" : true
545 }
546 {
547   "_id" : ObjectId("5f132aebd022deabe244f26e"),
548   "departureAirport" : "LHR",
549   "arrivalAirport" : "TXL",
550   "aircraft" : "Airbus A320",
551   "distance" : 950,
552   "intercontinental" : false
553 }
554
555
556 Delete
557 deleteOne(filter, options)
558 deleteMany(filter, options)
559
560 db.flightData.deleteOne({departureAirport : "MUC"})
561 > db.flightData.deleteMany({marker: "toDelete"})
562
563
564 ----- Projection-----
565
566 > db.passengers.find({}, {name: 1}).pretty()
567 {
568   "_id" : ObjectId("5f1339f7d022deabe244f26f"),
569   "name" : "Max Schwarzmüller"
570 }
571 { "_id" : ObjectId("5f1339f7d022deabe244f270"), "name" : "Manu Lorenz" }
572 { "_id" : ObjectId("5f1339f7d022deabe244f271"), "name" : "Chris Hayton" }
573 { "_id" : ObjectId("5f1339f7d022deabe244f272"), "name" : "Sandeep Kumar" }
574 { "_id" : ObjectId("5f1339f7d022deabe244f273"), "name" : "Maria Jones" }
575 { "_id" : ObjectId("5f1339f7d022deabe244f274"), "name" : "Alexandra Maier" }
576 { "_id" : ObjectId("5f1339f7d022deabe244f275"), "name" : "Dr. Phil Evans" }
577 { "_id" : ObjectId("5f1339f7d022deabe244f276"), "name" : "Sandra Brugge" }
578 { "_id" : ObjectId("5f1339f7d022deabe244f277"), "name" : "Elisabeth Mayr" }
579 { "_id" : ObjectId("5f1339f7d022deabe244f278"), "name" : "Frank Cube" }
580 { "_id" : ObjectId("5f1339f7d022deabe244f279"), "name" : "Karandeep Alun" }
581 { "_id" : ObjectId("5f1339f7d022deabe244f27a"), "name" : "Michaela Drayer" }
582 { "_id" : ObjectId("5f1339f7d022deabe244f27b"), "name" : "Bernd Hoftstadt" }
583 { "_id" : ObjectId("5f1339f7d022deabe244f27c"), "name" : "Scott Tolib" }
584 { "_id" : ObjectId("5f1339f7d022deabe244f27d"), "name" : "Freddy Melfer" }
585 { "_id" : ObjectId("5f1339f7d022deabe244f27e"), "name" : "Alexis Bohed" }
586 { "_id" : ObjectId("5f1339f7d022deabe244f27f"), "name" : "Melanie Palace" }
587 { "_id" : ObjectId("5f1339f7d022deabe244f280"), "name" : "Armin Glutch" }
588 { "_id" : ObjectId("5f1339f7d022deabe244f281"), "name" : "Klaus Arber" }
589 { "_id" : ObjectId("5f1339f7d022deabe244f282"), "name" : "Albert Twostone" }
590
591
592 > db.passengers.find({}, {name: 1, _id: 0}).pretty()
593 { "name" : "Max Schwarzmüller" }
594 { "name" : "Manu Lorenz" }
595 { "name" : "Chris Hayton" }
596 { "name" : "Sandeep Kumar" }
597 { "name" : "Maria Jones" }
598 { "name" : "Alexandra Maier" }
599 { "name" : "Dr. Phil Evans" }
600 { "name" : "Sandra Brugge" }
601 { "name" : "Elisabeth Mayr" }
602 { "name" : "Frank Cube" }
603 { "name" : "Karandeep Alun" }
604 { "name" : "Michaela Drayer" }

```

```

605 { "name" : "Bernd Hoftstadt" }
606 { "name" : "Scott Tolib" }
607 { "name" : "Freddy Melfer" }
608 { "name" : "Alexis Bohed" }
609 { "name" : "Melanie Palace" }
610 { "name" : "Armin Glutch" }
611 { "name" : "Klaus Arber" }
612 { "name" : "Albert Twostone" }
613
614 -----Embedded Documents-----
615
616 nested documents
617 have size limit limitations (50mb) up to 100 times
618
619 > db.flightData.updateMany({},{$set:{status:{description: "on-time", lastupdated:
620 "i Hour ago"}}})
621 { "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 0 }
622 > db.flightData.find().pretty()
623 {
624   "_id" : ObjectId("5f132aebd022deabe244f26d"),
625   "departureAirport" : "MUC",
626   "arrivalAirport" : "SFO",
627   "aircraft" : "Airbus A380",
628   "distance" : 12000,
629   "intercontinental" : true,
630   "status" : {
631     "description" : "on-time",
632     "lastupdated" : "i Hour ago"
633   }
634 }
635 {
636   "_id" : ObjectId("5f132aebd022deabe244f26e"),
637   "departureAirport" : "LHR",
638   "arrivalAirport" : "TXL",
639   "aircraft" : "Airbus A320",
640   "distance" : 950,
641   "intercontinental" : false,
642   "status" : {
643     "description" : "on-time",
644     "lastupdated" : "i Hour ago"
645   }
646 }
647 > db.flightData.updateMany({},{$set:{status:{description: "on-time", lastupdated:
648 "i Hour ago",details:{responsible: "RAJU"}}}}})
649 { "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
650 > db.flightData.find().pretty()
651 {
652   "_id" : ObjectId("5f132aebd022deabe244f26d"),
653   "departureAirport" : "MUC",
654   "arrivalAirport" : "SFO",
655   "aircraft" : "Airbus A380",
656   "distance" : 12000,
657   "intercontinental" : true,
658   "status" : {
659     "description" : "on-time",
660     "lastupdated" : "i Hour ago",
661     "details" : {
662       "responsible" : "RAJU"
663     }
664   }
665 }
666 {
667   "_id" : ObjectId("5f132aebd022deabe244f26e"),
668   "departureAirport" : "LHR",
669   "arrivalAirport" : "TXL",
670   "aircraft" : "Airbus A320",
671   "distance" : 950,
672   "intercontinental" : false,

```

```

672     "status" : {
673       "description" : "on-time",
674       "lastupdated" : "i Hour ago",
675       "details" : {
676         "responsible" : "RAJU"
677       }
678     }
679   }
680
681 // query an documents
682 > db.flightData.find({"status.description": "on-time"}).pretty()
683 {
684   "_id" : ObjectId("5f132aebd022deabe244f26d"),
685   "departureAirport" : "MUC",
686   "arrivalAirport" : "SFO",
687   "aircraft" : "Airbus A380",
688   "distance" : 12000,
689   "intercontinental" : true,
690   "status" : {
691     "description" : "on-time",
692     "lastupdated" : "i Hour ago",
693     "details" : {
694       "responsible" : "RAJU"
695     }
696   }
697 }
698 {
699   "_id" : ObjectId("5f132aebd022deabe244f26e"),
700   "departureAirport" : "LHR",
701   "arrivalAirport" : "TXL",
702   "aircraft" : "Airbus A320",
703   "distance" : 950,
704   "intercontinental" : false,
705   "status" : {
706     "description" : "on-time",
707     "lastupdated" : "i Hour ago",
708     "details" : {
709       "responsible" : "RAJU"
710     }
711   }
712 }
713
714

```

#### -----Arrays-----

Array of imbedded documents

Array can hold any data

```

719
720 > db.passengers.updateOne({name: "Albert Twostone"}, {$set:{hobbies:["sports",
721 "cooking"]}})
722 { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
723 > db.passengers.find().pretty()
724 {
725   "_id" : ObjectId("5f1339f7d022deabe244f26f"),
726   "name" : "Max SchwarzmueLLer",
727   "age" : 29
728 }
729 {
730   "_id" : ObjectId("5f1339f7d022deabe244f270"),
731   "name" : "Manu Lorenz",
732   "age" : 30
733 }
734 {
735   "_id" : ObjectId("5f1339f7d022deabe244f271"),
736   "name" : "Chris Hayton",
737   "age" : 35
738 }
739 {
740   "_id" : ObjectId("5f1339f7d022deabe244f272"),

```

```
740         "name" : "Sandeep Kumar",
741         "age" : 28
742     }
743     {
744         "_id" : ObjectId("5f1339f7d022deabe244f273") ,
745         "name" : "Maria Jones",
746         "age" : 30
747     }
748     {
749         "_id" : ObjectId("5f1339f7d022deabe244f274") ,
750         "name" : "Alexandra Maier",
751         "age" : 27
752     }
753     {
754         "_id" : ObjectId("5f1339f7d022deabe244f275") ,
755         "name" : "Dr. Phil Evans",
756         "age" : 47
757     }
758     {
759         "_id" : ObjectId("5f1339f7d022deabe244f276") ,
760         "name" : "Sandra Brugge",
761         "age" : 33
762     }
763     {
764         "_id" : ObjectId("5f1339f7d022deabe244f277") ,
765         "name" : "Elisabeth Mayr",
766         "age" : 29
767     }
768     {
769         "_id" : ObjectId("5f1339f7d022deabe244f278") ,
770         "name" : "Frank Cube",
771         "age" : 41
772     }
773     {
774         "_id" : ObjectId("5f1339f7d022deabe244f279") ,
775         "name" : "Karandeep Alun",
776         "age" : 48
777     }
778     {
779         "_id" : ObjectId("5f1339f7d022deabe244f27a") ,
780         "name" : "Michaela Drayer",
781         "age" : 39
782     }
783     {
784         "_id" : ObjectId("5f1339f7d022deabe244f27b") ,
785         "name" : "Bernd Hoftstadt",
786         "age" : 22
787     }
788     {
789         "_id" : ObjectId("5f1339f7d022deabe244f27c") ,
790         "name" : "Scott Tolib",
791         "age" : 44
792     }
793     {
794         "_id" : ObjectId("5f1339f7d022deabe244f27d") ,
795         "name" : "Freddy Melfer",
796         "age" : 41
797     }
798     {
799         "_id" : ObjectId("5f1339f7d022deabe244f27e") ,
800         "name" : "Alexis Bohed",
801         "age" : 35
802     }
803     {
804         "_id" : ObjectId("5f1339f7d022deabe244f27f") ,
805         "name" : "Melanie Palace",
806         "age" : 27
807     }
808     {
```

```

809     "_id" : ObjectId("5f1339f7d022deabe244f280"),
810     "name" : "Armin Glutch",
811     "age" : 35
812   }
813   {
814     "_id" : ObjectId("5f1339f7d022deabe244f281"),
815     "name" : "Klaus Arber",
816     "age" : 53
817   }
818   {
819     "_id" : ObjectId("5f1339f7d022deabe244f282"),
820     "name" : "Albert Twostone",
821     "age" : 68,
822     "hobbies" : [
823       "sports",
824       "cooking"
825     ]
826   }
827
828 > db.passengers.find({name: "Albert Twostone"}).pretty()
829 {
830   "_id" : ObjectId("5f1339f7d022deabe244f282"),
831   "name" : "Albert Twostone",
832   "age" : 68,
833   "hobbies" : [
834     "sports",
835     "cooking"
836   ]
837 }
838 > db.passengers.find({name: "Albert Twostone"}).hobbies
839 > db.passengers.findOne({name: "Albert Twostone"}).hobbies
840 [ "sports", "cooking" ]
841
842 // query an array
843 > db.passengers.find({hobbies: "sports"}).pretty()
844 {
845   "_id" : ObjectId("5f1339f7d022deabe244f282"),
846   "name" : "Albert Twostone",
847   "age" : 68,
848   "hobbies" : [
849     "sports",
850     "cooking"
851   ]
852 }

```

---

### 03-schemas-relations-how-to-structure-documents

---

```

860
861 > db.companies.insertOne({name: "Freash Apples Inc", isStartup: true, employees:
33, funding: 123456789876543219, details: {cea: "Mark Super"}, tags: [{title:
"super"}, {title: "perfect"}], foundingData: new Date(), insertedAt: new Timestamp()})
862 {
863   "acknowledged" : true,
864   "insertedId" : ObjectId("5f13ec400249b11a6aa5e37f")
865 }
866 > db.companies.find()
867 { "_id" : ObjectId("5f13ec400249b11a6aa5e37f"), "name" : "Freash Apples Inc",
"isStartup" : true, "employees" : 33, "funding" : 123456789876543220, "details" : {
"cea" : "Mark Super" }, "tags" : [ { "title" : "super" }, { "title" : "perfect" }
], "foundingData" : ISODate("2020-07-19T06:46:24.175Z"), "insertedAt" :
Timestamp(1595141184, 1) }
868 > db.companies.find().pretty()
869 {

```

```

870         "_id" : ObjectId("5f13ec400249b11a6aa5e37f"),
871         "name" : "Freash Apples Inc",
872         "isStartup" : true,
873         "employees" : 33,
874         "funding" : 123456789876543220,
875         "details" : {
876             "cea" : "Mark Super"
877         },
878         "tags" : [
879             {
880                 "title" : "super"
881             },
882             {
883                 "title" : "perfect"
884             }
885         ],
886         "foundingData" : ISODate("2020-07-19T06:46:24.175Z"),
887         "insertedAt" : Timestamp(1595141184, 1)
888     }
889
890 -----get the statistics of database-----
891 > db.numbers.insertOne({a: 1})
892 {
893     "acknowledged" : true,
894     "insertedId" : ObjectId("5f13ed5a0249b11a6aa5e380")
895 }
896 > db.numbers.findOne()
897 { "_id" : ObjectId("5f13ed5a0249b11a6aa5e380"), "a" : 1 }
898 > db.stats
899 function (scale) {
900     return this.runCommand({dbstats: 1, scale: scale});
901 }
902 > db.stats()
903 {
904     "db" : "companyData",
905     "collections" : 2,
906     "views" : 0,
907     "objects" : 2,
908     "avgObjSize" : 135,
909     "dataSize" : 270,
910     "storageSize" : 20480,
911     "numExtents" : 0,
912     "indexes" : 2,
913     "indexSize" : 20480,
914     "fsUsedSize" : 54183743488,
915     "fsTotalSize" : 61754699776,
916     "ok" : 1
917 }
918
919 // differentiate data size
920 > db.numbers.insertOne({a: 1})
921 {
922     "acknowledged" : true,
923     "insertedId" : ObjectId("5f13eee00249b11a6aa5e381")
924 }
925 > db.stats()
926 {
927     "db" : "companyData",
928     "collections" : 1,
929     "views" : 0,
930     "objects" : 1,
931     "avgObjSize" : 33,
932     "dataSize" : 33,
933     "storageSize" : 4096,
934     "numExtents" : 0,
935     "indexes" : 1,
936     "indexSize" : 4096,
937     "fsUsedSize" : 54183714816,
938     "fsTotalSize" : 61754699776,

```

```

939         "ok" : 1
940     }
941
942     > db.numbers.drop()
943     true
944     > db.numbers.insertOne({a: NumberInt(1)})
945     {
946         "acknowledged" : true,
947         "insertedId" : ObjectId("5f13ef440249b11a6aa5e382")
948     }
949     > db.stats()
950     {
951         "db" : "companyData",
952         "collections" : 1,
953         "views" : 0,
954         "objects" : 1,
955         "avgObjSize" : 29,
956         "dataSize" : 29,
957         "storageSize" : 4096,
958         "numExtents" : 0,
959         "indexes" : 1,
960         "indexSize" : 4096,
961         "fsUsedSize" : 54183739392,
962         "fsTotalSize" : 61754699776,
963         "ok" : 1
964     }
965
966     -----getting the datatype-----
967
968     > db.numbers.insertOne({a: 1.5,b: "r"})
969     {
970         "acknowledged" : true,
971         "insertedId" : ObjectId("5f13f19e0249b11a6aa5e386")
972     }
973     > typeof db.numbers.findOne({b: "r"}).a
974     number
975
976     -----Relations-----
977     one-to-one
978     one-to-many
979     many-to-many
980
981     -----Joining with $lookup-----
982
983     > use bookStore
984     switched to db bookStore
985     > cls
986
987     > db.authors.insertMany([{name: 'Max Schwarz',age: 29, address:{street:
988     'Main'}}},{name: 'Manuel Lor',age: 30, address:{street: 'Tree'}}])
989     {
990         "acknowledged" : true,
991         "insertedIds" : [
992             ObjectId("5f145a7c231893e15e9e53fe"),
993             ObjectId("5f145a7c231893e15e9e53ff")
994         ]
995     }
996     > db.authors.find().pretty()
997     {
998         "_id" : ObjectId("5f145a7c231893e15e9e53fe"),
999         "name" : "Max Schwarz",
1000         "age" : 29,
1001         "address" : {
1002             "street" : "Main"
1003         }
1004     }
1005     {
1006         "_id" : ObjectId("5f145a7c231893e15e9e53ff"),
1007         "name" : "Manuel Lor",

```

```

1007         "age" : 30,
1008         "address" : {
1009             "street" : "Tree"
1010         }
1011     }
1012 > db.books.insertOne({name: 'My favorite
Book', authors: [ObjectId("5f145a7c231893e15e9e53fe"), ObjectId("5f145a7c231893e15e9e53f
f")]}))
1013 {
1014     "acknowledged" : true,
1015     "insertedId" : ObjectId("5f145b5f231893e15e9e5400")
1016 }
1017 > db.authors.findOne()
1018 {
1019     "_id" : ObjectId("5f145a7c231893e15e9e53fe"),
1020     "name" : "Max Schwarz",
1021     "age" : 29,
1022     "address" : {
1023         "street" : "Main"
1024     }
1025 }
1026 > db.books.findOne()
1027 {
1028     "_id" : ObjectId("5f145b5f231893e15e9e5400"),
1029     "name" : "My favorite Book",
1030     "authors" : [
1031         ObjectId("5f145a7c231893e15e9e53fe"),
1032         ObjectId("5f145a7c231893e15e9e53ff")
1033     ]
1034 }
1035
1036
1037 > db.books.aggregate([{$lookup:{from:
'authors',localField:'authors',foreignField:"_id", as:'creators'}}])
1038 { "_id" : ObjectId("5f145b5f231893e15e9e5400"), "name" : "My favorite Book",
"authors" : [ ObjectId("5f145a7c231893e15e9e53fe"),
ObjectId("5f145a7c231893e15e9e53ff") ], "creators" : [ { "_id" :
ObjectId("5f145a7c231893e15e9e53fe"), "name" : "Max Schwarz", "age" : 29, "address"
: { "street" : "Main" } }, { "_id" : ObjectId("5f145a7c231893e15e9e53ff"), "name" :
"Manuel Lor", "age" : 30, "address" : { "street" : "Tree" } } ] }
1039 > db.books.aggregate([{$lookup:{from:
'authors',localField:'authors',foreignField:"_id", as:'creators'}}]).pretty()
1040 {
1041     "_id" : ObjectId("5f145b5f231893e15e9e5400"),
1042     "name" : "My favorite Book",
1043     "authors" : [
1044         ObjectId("5f145a7c231893e15e9e53fe"),
1045         ObjectId("5f145a7c231893e15e9e53ff")
1046     ],
1047     "creators" : [
1048         {
1049             "_id" : ObjectId("5f145a7c231893e15e9e53fe"),
1050             "name" : "Max Schwarz",
1051             "age" : 29,
1052             "address" : {
1053                 "street" : "Main"
1054             }
1055         },
1056         {
1057             "_id" : ObjectId("5f145a7c231893e15e9e53ff"),
1058             "name" : "Manuel Lor",
1059             "age" : 30,
1060             "address" : {
1061                 "street" : "Tree"
1062             }
1063         }
1064     ]
1065 }
1066

```



```

1067 > db.books.find().pretty()
1068 {
1069   "_id" : ObjectId("5f145b5f231893e15e9e5400"),
1070   "name" : "My favorite Book",
1071   "authors" : [
1072     ObjectId("5f145a7c231893e15e9e53fe"),
1073     ObjectId("5f145a7c231893e15e9e53ff")
1074   ]
1075 }
1076
1077
1078
1079
1080 -----
1081 04-simple-project-blog
1082 -----
1083
1084 -----Blog Project-----
1085
1086 > use blog
1087 switched to db blog
1088 > db.users.insertMany([{name: 'Max Schwaezmuller', age:
29,email:'max23@gmail.com'},{name: 'Raju', age:'22', email: 'mijanur231@gmail.com'}])
1089 {
1090   "acknowledged" : true,
1091   "insertedIds" : [
1092     ObjectId("5f146535231893e15e9e5401"),
1093     ObjectId("5f146535231893e15e9e5402")
1094   ]
1095 }
1096 > db.users.find().pretty()
1097 {
1098   "_id" : ObjectId("5f146535231893e15e9e5401"),
1099   "name" : "Max Schwaezmuller",
1100   "age" : 29,
1101   "email" : "max23@gmail.com"
1102 }
1103 {
1104   "_id" : ObjectId("5f146535231893e15e9e5402"),
1105   "name" : "Raju",
1106   "age" : "22",
1107   "email" : "mijanur231@gmail.com"
1108 }
1109 > db.post.insertOne({title: 'My first post is', text:'This is very important post,
I hope you like it!',tags:['new', 'tech'], creator:
ObjectId("5f146535231893e15e9e5402"), comments: [{text: 'I like this post', author:
ObjectId("5f146535231893e15e9e5401")}]})
1110 {
1111   "acknowledged" : true,
1112   "insertedId" : ObjectId("5f146bba231893e15e9e5403")
1113 }
1114 > db.posts.findOne()
1115 null
1116 > db.post.findOne()
1117 {
1118   "_id" : ObjectId("5f146bba231893e15e9e5403"),
1119   "title" : "My first post is",
1120   "text" : "This is very important post, I hope you like it!",
1121   "tags" : [
1122     "new",
1123     "tech"
1124   ],
1125   "creator" : ObjectId("5f146535231893e15e9e5402"),
1126   "comments" : [
1127     {
1128       "text" : "I like this post",
1129       "author" : ObjectId("5f146535231893e15e9e5401")

```

```

1130     }
1131   ]
1132 }
1133
1134 -----validation check-----
1135
1136 > db.post.drop()
1137 true
1138 > db.post.findOne()
1139 null
1140 > db.createCollection('post', {
1141   ...   validator: {
1142   ...     $jsonSchema: {
1143   ...       bsonType: 'object',
1144   ...       required: ['title', 'text', 'creator', 'comments'],
1145   ...       properties: {
1146   ...         title: {
1147   ...           bsonType: 'string',
1148   ...           description: 'must be a string and is required'
1149   ...         },
1150   ...         text: {
1151   ...           bsonType: 'string',
1152   ...           description: 'must be a string and is required'
1153   ...         },
1154   ...         creator: {
1155   ...           bsonType: 'objectId',
1156   ...           description: 'must be an objectid and is required'
1157   ...         },
1158   ...         comments: {
1159   ...           bsonType: 'array',
1160   ...           description: 'must be an array and is required',
1161   ...           items: {
1162   ...             bsonType: 'object',
1163   ...             required: ['text', 'author'],
1164   ...             properties: {
1165   ...               text: {
1166   ...                 bsonType: 'string',
1167   ...                 description: 'must be a string and is required'
1168   ...               },
1169   ...               author: {
1170   ...                 bsonType: 'objectId',
1171   ...                 description: 'must be an objectid and is required'
1172   ...               }
1173   ...             }
1174   ...           }
1175   ...         }
1176   ...       }
1177   ...     }
1178   ...   });
1179 { "ok" : 1 }
1180
1181 > db.post.insertOne({title: 'My first post is', text:'This is very important post,
I hope you like it!',tags:['new', 'tech'], creator:
ObjectId("5f146535231893e15e9e5402"), comments: [{text: 'I like this post', author:
ObjectId("5f146535231893e15e9e5401")}]}))
1182 {
1183   "acknowledged" : true,
1184   "insertedId" : ObjectId("5f148c00231893e15e9e5404")
1185 }
1186 > db.post.findOne()
1187 {
1188   "_id" : ObjectId("5f148c00231893e15e9e5404"),
1189   "title" : "My first post is",
1190   "text" : "This is very important post, I hope you like it!",
1191   "tags" : [
1192     "new",
1193     "tech"
1194   ],
1195   "creator" : ObjectId("5f146535231893e15e9e5402"),

```

```

1196     "comments" : [
1197         {
1198             "text" : "I like this post",
1199             "author" : ObjectId("5f146535231893e15e9e5401")
1200         }
1201     ]
1202 }
1203
1204
1205 > db.post.insertOne({title: 'My first post is', text:'This is very important post,
I hope you like it!',tags:['new', 'tech'], creator:
ObjectId("5f146535231893e15e9e5402"), comments: [{text: 'I like this post',
author:1234}]})
1206 2020-07-20T00:12:38.402+0600 E QUERY    [thread1] WriteError: Document failed
validation :
1207 WriteError({
1208     "index" : 0,
1209     "code" : 121,
1210     "errmsg" : "Document failed validation",
1211     "op" : {
1212         "_id" : ObjectId("5f148d16231893e15e9e5406"),
1213         "title" : "My first post is",
1214         "text" : "This is very important post, I hope you like it!",
1215         "tags" : [
1216             "new",
1217             "tech"
1218         ],
1219         "creator" : ObjectId("5f146535231893e15e9e5402"),
1220         "comments" : [
1221             {
1222                 "text" : "I like this post",
1223                 "author" : 1234
1224             }
1225         ]
1226     }
1227 })
1228 WriteError@src/mongo/shell/bulk_api.js:466:48
1229 Bulk/mergeBatchResults@src/mongo/shell/bulk_api.js:846:49
1230 Bulk/executeBatch@src/mongo/shell/bulk_api.js:910:13
1231 Bulk/this.execute@src/mongo/shell/bulk_api.js:1154:21
1232 DBCollection.prototype.insertOne@src/mongo/shell/crud_api.js:252:9
1233 @(shell):1:1
1234
1235 > db.post.find().pretty()
1236 {
1237     "_id" : ObjectId("5f148d87231893e15e9e5407"),
1238     "title" : "My first post is",
1239     "text" : "This is very important post, I hope you like it!",
1240     "tags" : [
1241         "new",
1242         "tech"
1243     ],
1244     "creator" : ObjectId("5f146535231893e15e9e5402"),
1245     "comments" : [
1246         {
1247             "text" : "I like this post",
1248             "author" : ObjectId("5f146535231893e15e9e5401")
1249         }
1250     ]
1251 }
1252
1253 -----administrative check-----
1254 // collMod -> collections mode
1255
1256 > db.runCommand({collMod: 'post', validator: {
1257     ...   $jsonSchema: {
1258     ...     bsonType: 'object',
1259     ...     required: ['title', 'text', 'creator', 'comments'],
1260     ...     properties: {

```

```

1261     ...     title: {
1262     ...         bsonType: 'string',
1263     ...         description: 'must be a string and is required'
1264     ...     },
1265     ...     text: {
1266     ...         bsonType: 'string',
1267     ...         description: 'must be a string and is required'
1268     ...     },
1269     ...     creator: {
1270     ...         bsonType: 'objectId',
1271     ...         description: 'must be an objectid and is required'
1272     ...     },
1273     ...     comments: {
1274     ...         bsonType: 'array',
1275     ...         description: 'must be an array and is required',
1276     ...         items: {
1277     ...             bsonType: 'object',
1278     ...             required: ['text', 'author'],
1279     ...             properties: {
1280     ...                 text: {
1281     ...                     bsonType: 'string',
1282     ...                     description: 'must be a string and is required'
1283     ...                 },
1284     ...                 author: {
1285     ...                     bsonType: 'objectId',
1286     ...                     description: 'must be an objectid and is required'
1287     ...                 }
1288     ...             }
1289     ...         }
1290     ...     }
1291     ... }
1292     ... }
1293     ... )))
1294 { "ok" : 1 }

```

```

1295
1296
1297 > db.runCommand({
1298     collMod: 'posts',
1299     validator: {
1300         $jsonSchema: {
1301             bsonType: 'object',
1302             required: ['title', 'text', 'creator', 'comments'],
1303             properties: {
1304                 title: {
1305                     bsonType: 'string',
1306                     description: 'must be a string and is required'
1307                 },
1308                 text: {
1309                     bsonType: 'string',
1310                     description: 'must be a string and is required'
1311                 },
1312                 creator: {
1313                     bsonType: 'objectId',
1314                     description: 'must be an objectid and is required'
1315                 },
1316                 comments: {
1317                     bsonType: 'array',
1318                     description: 'must be an array and is required',
1319                     items: {
1320                         bsonType: 'object',
1321                         required: ['text', 'author'],
1322                         properties: {
1323                             text: {
1324                                 bsonType: 'string',
1325                                 description: 'must be a string and is required'
1326                             },
1327                             author: {
1328                                 bsonType: 'objectId',
1329                                 description: 'must be an objectid and is required'

```

```

1330         }
1331     }
1332 }
1333 }
1334 }
1335 }
1336 },
1337 validationAction: 'warn'
1338 });
1339 { "ok" : 1 }
1340 > db.post.insertOne({title: 'My first post is', text:'This is very important post,
I hope you like it!',tags:['new', 'tech'], creator:
ObjectId("5f146535231893e15e9e5402"), comments: [{text: 'I like this post',
author:1234}]})
1341 {
1342     "acknowledged" : true,
1343     "insertedId" : ObjectId("5f14930f231893e15e9e5409")
1344 }
1345
1346
1347
1348
1349 -----
-----
1350 05-shell-important-commands
1351 -----
-----
1352
1353
1354 mongod --help
1355
1356 > help
1357     db.help()                help on db methods
1358     db.mycoll.help()         help on collection methods
1359     sh.help()                sharding helpers
1360     rs.help()                replica set helpers
1361     help admin               administrative help
1362     help connect             connecting to a db help
1363     help keys                key shortcuts
1364     help misc                misc things to know
1365     help mr                  mapreduce
1366
1367     show dbs                 show database names
1368     show collections          show collections in current database
1369     show users                show users in current database
1370     show profile              show most recent system.profile entries with time
>= 1ms
1371     show logs                 show the accessible logger names
1372     show log [name]          prints out the last segment of log in memory,
'global' is default
1373     use <db_name>            set current database
1374     db.foo.find()             list objects in collection foo
1375     db.foo.find( { a : 1 } ) list objects in foo where a == 1
1376     it                        result of the last line evaluated; use to further
iterate
1377     DBQuery.shellBatchSize = x set default number of items to display on shell
1378     exit                     quit the mongo shell
1379
1380
1381
1382 > use shop
1383 switched to db shop
1384 > db.help()
1385 DB methods:
1386     db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs command
[just calls db.runCommand(...)]
1387     db.aggregate([pipeline], {options}) - performs a collectionless aggregation on
this database; returns a cursor
1388     db.auth(username, password)

```

```

1389 db.cloneDatabase(fromhost)
1390 db.commandHelp(name) returns the help for the command
1391 db.copyDatabase(fromdb, todb, fromhost)
1392 db.createCollection(name, {size: ..., capped: ..., max: ...})
1393 db.createView(name, viewOn, [{operator: {...}}, ...], {viewOptions})
1394 db.createUser(userDocument)
1395 db.currentOp() displays currently executing operations in the db
1396 db.dropDatabase()
1397 db.eval() - deprecated
1398 db.fsyncLock() flush data to disk and lock server for backups
1399 db.fsyncUnlock() unlocks server following a db.fsyncLock()
1400 db.getCollection(cname) same as db['cname'] or db.cname
1401 db.getCollectionInfos([filter]) - returns a list that contains the names and
options of the db's collections
1402 db.getCollectionNames()
1403 db.getLastErrorMessage() - just returns the err msg string
1404 db.getLastErrorMessageObj() - return full status object
1405 db.getLogComponents()
1406 db.getMongo() get the server connection object
1407 db.getMongo().setSlaveOk() allow queries on a replication slave server
1408 db.getName()
1409 db.getPrevError()
1410 db.getProfilingLevel() - deprecated
1411 db.getProfilingStatus() - returns if profiling is on and slow threshold
1412 db.getReplicationInfo()
1413 db.getSiblingDB(name) get the db at the same server as this one
1414 db.getWriteConcern() - returns the write concern used for any operations on
this db, inherited from server object if set
1415 db.hostInfo() get details about the server's host
1416 db.isMaster() check replica primary status
1417 db.killOp(opid) kills the current operation in the db
1418 db.listCommands() lists all the db commands
1419 db.loadServerScripts() loads all the scripts in db.system.js
1420 db.logout()
1421 db.printCollectionStats()
1422 db.printReplicationInfo()
1423 db.printShardingStatus()
1424 db.printSlaveReplicationInfo()
1425 db.dropUser(username)
1426 db.repairDatabase()
1427 db.resetError()
1428 db.runCommand(cmdObj) run a database command. if cmdObj is a string, turns it
into {cmdObj: 1}
1429 db.serverStatus()
1430 db.setLogLevel(level,<component>)
1431 db.setProfilingLevel(level,slowms) 0=off 1=slow 2=all
1432 db.setWriteConcern(<write concern doc>) - sets the write concern for writes to
the db
1433 db.unsetWriteConcern(<write concern doc>) - unsets the write concern for writes
to the db
1434 db.setVerboseShell(flag) display extra information in shell output
1435 db.shutdownServer()
1436 db.stats()
1437 db.version() current version of the server
1438
1439
1440
1441
1442 > show collections
1443 products
1444 > db.products.help()
1445 DBCollection help
1446 db.products.find().help() - show DBCursor help
1447 db.products.bulkWrite( operations, <optional params> ) - bulk execute write
operations, optional parameters are: w, wtimeout, j
1448 db.products.count( query = {}, <optional params> ) - count the number of
documents that matches the query, optional parameters are: limit, skip, hint,
maxTimeMS
1449 db.products.copyTo(newColl) - duplicates collection by copying all documents to

```

```

newColl; no indexes are copied.
1450 db.products.convertToCapped(maxBytes) - calls {convertToCapped:'products',
size:maxBytes}} command
1451 db.products.createIndex(keypattern[,options])
1452 db.products.createIndexes([keypatterns], <options>)
1453 db.products.dataSize()
1454 db.products.deleteOne( filter, <optional params> ) - delete first matching
document, optional parameters are: w, wtimeout, j
1455 db.products.deleteMany( filter, <optional params> ) - delete all matching
documents, optional parameters are: w, wtimeout, j
1456 db.products.distinct( key, query, <optional params> ) - e.g.
db.products.distinct( 'x' ), optional parameters are: maxTimeMS
1457 db.products.drop() drop the collection
1458 db.products.dropIndex(index) - e.g. db.products.dropIndex( "indexName" ) or
db.products.dropIndex( { "indexKey" : 1 } )
1459 db.products.dropIndexes()
1460 db.products.ensureIndex(keypattern[,options]) - DEPRECATED, use createIndex()
instead
1461 db.products.explain().help() - show explain help
1462 db.products.reIndex()
1463 db.products.find([query],[fields]) - query is an optional query filter. fields
is optional set of fields to return.
1464 e.g. db.products.find( {x:77} ,
{name:1, x:1} )
1465 db.products.find(...).count()
1466 db.products.find(...).limit(n)
1467 db.products.find(...).skip(n)
1468 db.products.find(...).sort(...)
1469 db.products.findOne([query], [fields], [options], [readConcern])
1470 db.products.findOneAndDelete( filter, <optional params> ) - delete first
matching document, optional parameters are: projection, sort, maxTimeMS
1471 db.products.findOneAndReplace( filter, replacement, <optional params> ) -
replace first matching document, optional parameters are: projection, sort,
maxTimeMS, upsert, returnNewDocument
1472 db.products.findOneAndUpdate( filter, update, <optional params> ) - update
first matching document, optional parameters are: projection, sort, maxTimeMS,
upsert, returnNewDocument
1473 db.products.getDB() get DB object associated with collection
1474 db.products.getPlanCache() get query plan cache associated with collection
1475 db.products.getIndexes()
1476 db.products.group( { key : ..., initial: ..., reduce : ...[, cond: ...] } )
1477 db.products.insert(obj)
1478 db.products.insertOne( obj, <optional params> ) - insert a document, optional
parameters are: w, wtimeout, j
1479 db.products.insertMany( [objects], <optional params> ) - insert multiple
documents, optional parameters are: w, wtimeout, j
1480 db.products.mapReduce( mapFunction , reduceFunction , <optional params> )
1481 db.products.aggregate( [pipeline], <optional params> ) - performs an
aggregation on a collection; returns a cursor
1482 db.products.remove(query)
1483 db.products.replaceOne( filter, replacement, <optional params> ) - replace the
first matching document, optional parameters are: upsert, w, wtimeout, j
1484 db.products.renameCollection( newName , <dropTarget> ) renames the collection.
1485 db.products.runCommand( name , <options> ) runs a db command with the given
name where the first param is the collection name
1486 db.products.save(obj)
1487 db.products.stats({scale: N, indexDetails: true/false, indexDetailsKey: <index
key>, indexDetailsName: <index name>})
1488 db.products.storageSize() - includes free space allocated to this collection
1489 db.products.totalIndexSize() - size in bytes of all the indexes
1490 db.products.totalSize() - storage allocated for all data and indexes
1491 db.products.update( query, object[, upsert_bool, multi_bool] ) - instead of two
flags, you can pass an object with fields: upsert, multi
1492 db.products.updateOne( filter, update, <optional params> ) - update the first
matching document, optional parameters are: upsert, w, wtimeout, j
1493 db.products.updateMany( filter, update, <optional params> ) - update all
matching documents, optional parameters are: upsert, w, wtimeout, j
1494 db.products.validate( <full> ) - SLOW
1495 db.products.getShardVersion() - only for use with sharding

```

```

1496 db.products.getShardDistribution() - prints statistics about data distribution
    in the cluster
1497 db.products.getSplitKeysForChunks( <maxChunkSize> ) - calculates split points
    over all chunks and returns splitter function
1498 db.products.getWriteConcern() - returns the write concern used for any
    operations on this collection, inherited from server/db if set
1499 db.products.setWriteConcern( <write concern doc> ) - sets the write concern for
    writes to the collection
1500 db.products.unsetWriteConcern( <write concern doc> ) - unsets the write concern
    for writes to the collection
1501 db.products.latencyStats() - display operation latency histograms for this
    collection

```

```

1502
1503
1504
1505
1506

```

```

-----
-----

```

```

1507 06-crud-operations-advanced

```

```

1508

```

```

-----
-----

```

```

1509

```

```

1510 -----CREATE-----

```

```

1511

```

```

1512 insert() --> insert also exist one many document.But it's not recommended to use
    it anymore - it also does not return the inserted id's

```

```

1513

```

```

1514 > use user

```

```

1515 switched to db user

```

```

1516 > db.persons.insert({name: 'Phil', age: 35})

```

```

1517 WriteResult({ "nInserted" : 1 })

```

```

1518 > db.persons.find()

```

```

1519 { "_id" : ObjectId("5f151e97e3242ab6a2f87b4e"), "name" : "Phil", "age" : 35 }

```

```

1520 > db.persons.find().pretty()

```

```

1521 {

```

```

1522   " _id" : ObjectId("5f151e97e3242ab6a2f87b4e"),

```

```

1523   "name" : "Phil",

```

```

1524   "age" : 35

```

```

1525 }

```

```

1526

```

```

1527

```

```

1528 > db.persons.insert([{name: 'Khil', age: 45},{name: 'RAJU', age: 22}])

```

```

1529 BulkWriteResult({

```

```

1530   "writeErrors" : [ ],

```

```

1531   "writeConcernErrors" : [ ],

```

```

1532   "nInserted" : 2,

```

```

1533   "nUpserted" : 0,

```

```

1534   "nMatched" : 0,

```

```

1535   "nModified" : 0,

```

```

1536   "nRemoved" : 0,

```

```

1537   "upserted" : [ ]

```

```

1538 })

```

```

1539

```

```

1540 > db.persons.find().pretty()

```

```

1541 {

```

```

1542   " _id" : ObjectId("5f151e97e3242ab6a2f87b4e"),

```

```

1543   "name" : "Phil",

```

```

1544   "age" : 35

```

```

1545 }

```

```

1546 {

```

```

1547   " _id" : ObjectId("5f15209ae3242ab6a2f87b4f"),

```

```

1548   "name" : "Khil",

```

```

1549   "age" : 45

```

```

1550 }

```

```

1551 {

```

```

1552   " _id" : ObjectId("5f15209ae3242ab6a2f87b50"),

```

```

1553   "name" : "RAJU",

```

```

1554   "age" : 22

```

```

1555 }

```



```

1556
1557
1558 -----working with order insert-----
1559
1560
1561 1. By default, when using insertMany(), inserts are ordered, that means, that the
    inserting process stops if an error occurs.
1562 2. Can changes this by switching to 'unordered inserts' - inserting process will
    then continue, even if errors occurred.
1563 3. In both cases, no successful inserts (before the error) will be rolled back.
1564 4. Successful insert will not roll back.
1565
1566 // bulk process
1567
1568
1569
1570 > db.hobbies.insertMany([{_id: 'sports', name:
'Sports'},{_id:'cooking',name:'Cooking'},{_id:'cars',name: 'Cars'}])
1571 {
1572   "acknowledged" : true,
1573   "insertedIds" : [
1574     "sports",
1575     "cooking",
1576     "cars"
1577   ]
1578 }
1579 > db.hobbies.find().pretty()
1580 { "_id" : "sports", "name" : "Sports" }
1581 { "_id" : "cooking", "name" : "Cooking" }
1582 { "_id" : "cars", "name" : "Cars" }
1583
1584 > db.hobbies.insertMany([{_id: 'yago', name: 'Yoga'},{_id:'cooking',name:'Cooking'}])
1585 2020-07-20T11:19:04.791+0600 E QUERY [thread1] BulkWriteError: write error at
item 1 in bulk operation :
BulkWriteError({
1586   "writeErrors" : [
1587     {
1588       "index" : 1,
1589       "code" : 11000,
1590       "errmsg" : "E11000 duplicate key error collection: user.hobbies index:
1591         _id_dup key: { : \"cooking\" }",
1592       "op" : {
1593         "_id" : "cooking",
1594         "name" : "Cooking"
1595       }
1596     }
1597   ],
1598   "writeConcernErrors" : [ ],
1599   "nInserted" : 1,
1600   "nUpserted" : 0,
1601   "nMatched" : 0,
1602   "nModified" : 0,
1603   "nRemoved" : 0,
1604   "upserted" : [ ]
1605 })
1606 BulkWriteError@src/mongo/shell/bulk_api.js:369:48
1607 BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:333:24
1608 Bulk/this.execute@src/mongo/shell/bulk_api.js:1177:1
1609 DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:314:5
1610 @(shell):1:1
1611
1612 // every element is inserted and standalone but if an error creates then exact
    element and after are is exited.
1613
1614 > db.hobbies.find().pretty()
1615 { "_id" : "sports", "name" : "Sports" }
1616 { "_id" : "cooking", "name" : "Cooking" }
1617 { "_id" : "cars", "name" : "Cars" }
1618 { "_id" : "yago", "name" : "Yoga" }

```

```

1619
1620
1621 > db.hobbies.insertMany([{_id: 'yago', name:
1622 'Yoga'}},{_id:'cooking',name:'Cooking'}},{_id:'hiking',name:'Hiking'}]},{ordered: true})
2020-07-20T11:33:12.922+0600 E QUERY [thread1] BulkWriteError: write error at
item 0 in bulk operation :
1623 BulkWriteError({
1624   "writeErrors" : [
1625     {
1626       "index" : 0,
1627       "code" : 11000,
1628       "errmsg" : "E11000 duplicate key error collection: user.hobbies index:
_id_dup key: { : \"yago\" }",
1629       "op" : {
1630         "_id" : "yago",
1631         "name" : "Yoga"
1632       }
1633     }
1634   ],
1635   "writeConcernErrors" : [ ],
1636   "nInserted" : 0,
1637   "nUpserted" : 0,
1638   "nMatched" : 0,
1639   "nModified" : 0,
1640   "nRemoved" : 0,
1641   "upserted" : [ ]
1642 })
1643 BulkWriteError@src/mongo/shell/bulk_api.js:369:48
1644 BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:333:24
1645 Bulk/this.execute@src/mongo/shell/bulk_api.js:1177:1
1646 DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:314:5
1647 @(shell):1:1
1648
1649 // now one element is inserted
1650
1651 > db.hobbies.insertMany([{_id: 'yago', name:
1652 'Yoga'}},{_id:'cooking',name:'Cooking'}},{_id:'hiking',name:'Hiking'}]},{ordered:
false})
2020-07-20T11:33:46.532+0600 E QUERY [thread1] BulkWriteError: 2 write errors in
bulk operation :
1653 BulkWriteError({
1654   "writeErrors" : [
1655     {
1656       "index" : 0,
1657       "code" : 11000,
1658       "errmsg" : "E11000 duplicate key error collection: user.hobbies index:
_id_dup key: { : \"yago\" }",
1659       "op" : {
1660         "_id" : "yago",
1661         "name" : "Yoga"
1662       }
1663     },
1664     {
1665       "index" : 1,
1666       "code" : 11000,
1667       "errmsg" : "E11000 duplicate key error collection: user.hobbies index:
_id_dup key: { : \"cooking\" }",
1668       "op" : {
1669         "_id" : "cooking",
1670         "name" : "Cooking"
1671       }
1672     }
1673   ],
1674   "writeConcernErrors" : [ ],
1675   "nInserted" : 1,
1676   "nUpserted" : 0,
1677   "nMatched" : 0,
1678   "nModified" : 0,
1679   "nRemoved" : 0,

```

```

1680         "upserted" : [ ]
1681     })
1682 BulkWriteError@src/mongo/shell/bulk_api.js:369:48
1683 BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:333:24
1684 Bulk/this.execute@src/mongo/shell/bulk_api.js:1177:1
1685 DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:314:5
1686 @(shell):1:1
1687
1688 > db.hobbies.find().pretty()
1689 { "_id" : "sports", "name" : "Sports" }
1690 { "_id" : "cooking", "name" : "Cooking" }
1691 { "_id" : "cars", "name" : "Cars" }
1692 { "_id" : "yago", "name" : "Yoga" }
1693 { "_id" : "hiking", "name" : "Hiking" }
1694
1695 -----Write concern-----
1696
1697 ## Control the "level of guarantee"
1698
1699 client---> MongoDB Server ---> Storage Engine ---> 1. Memory 2.Data on Disk
1700
1701 e.g insertOne() | --> {w: 1, j: undefined}
1702 | --> {w: 1, j: true} --> greater security that this will happen
1703 | and succeed even if the server should face
1704 | issues
1705 | --> {w: 1, wtimeout: 200, j: true} --> this simply means which
1706 | time frame do you give your
1707
1708 server to report a success
1709 for this write before you
1710 cancel it
1711
1712 w--> write -> write : 1 means should accepted to write
1713 ## In write the number means how many instances you want this write to be
1714 acknowledged. With 1 is the default. So the storage engine is aware of it and will
1715 eventually write to the disk.
1716
1717 j--> Journal('Todos') --> the journal is an additional file which the storage
1718 engine manages is like a To-Do file. It works when if server is down for some
1719 reason then file is still there.If the restart the server or if it recovers
1720 basically.
1721
1722 ## Backup todo list if server is down
1723
1724 > db.persons.insertOne({name: 'Chrissy', age: 44},{ writeConcern: {w: 0} })
1725 { "acknowledged" : false }
1726
1727 > db.persons.find()
1728 { "_id" : ObjectId("5f151e97e3242ab6a2f87b4e"), "name" : "Phil", "age" : 35 }
1729 { "_id" : ObjectId("5f15209ae3242ab6a2f87b4f"), "name" : "Khil", "age" : 45 }
1730 { "_id" : ObjectId("5f15209ae3242ab6a2f87b50"), "name" : "RAJU", "age" : 22 }
1731 { "_id" : ObjectId("5f154012e3242ab6a2f87b52"), "name" : "Chrissy", "age" : 44 }
1732
1733 // data is stored but acknowledged is false.You sent the request but you don't know
1734 if it reached the server.If any network connections issue create. W:0 is super fast
1735 but obviously, it tells you nothing about whether this succeed or not.
1736
1737 // write : the default is true
1738
1739 > db.persons.insertOne({name: 'Alex', age: 35},{writeConcern: {w: 1}})
1740 {
1741     "acknowledged" : true,
1742     "insertedId" : ObjectId("5f15415de3242ab6a2f87b53")
1743 }
1744
1745 > db.persons.find()
1746 { "_id" : ObjectId("5f151e97e3242ab6a2f87b4e"), "name" : "Phil", "age" : 35 }
1747 { "_id" : ObjectId("5f15209ae3242ab6a2f87b4f"), "name" : "Khil", "age" : 45 }
1748 { "_id" : ObjectId("5f15209ae3242ab6a2f87b50"), "name" : "RAJU", "age" : 22 }

```

```

1739 { "_id" : ObjectId("5f154012e3242ab6a2f87b52"), "name" : "Chrissy", "age" : 44 }
1740 { "_id" : ObjectId("5f15415de3242ab6a2f87b53"), "name" : "Alex", "age" : 35 }
1741
1742 // journal : default is false or undefined
1743
1744 > db.persons.insertOne({name: 'Michel', age: 35},{writeConcern: {w: 1, j: false}})
1745 {
1746   "acknowledged" : true,
1747   "insertedId" : ObjectId("5f154215e3242ab6a2f87b54")
1748 }
1749
1750 // if journal is true then it could be little bit slower
1751
1752 > db.persons.insertOne({name: 'Michela', age: 35},{writeConcern: {w: 1, j: true}})
1753 {
1754   "acknowledged" : true,
1755   "insertedId" : ObjectId("5f154255e3242ab6a2f87b55")
1756 }
1757
1758 > db.persons.insertOne({name: 'Aliya', age: 35},{writeConcern: {w: 1, j:
true,wtimeout: 200}})
1759 {
1760   "acknowledged" : true,
1761   "insertedId" : ObjectId("5f1542cee3242ab6a2f87b56")
1762 }
1763
1764 // it super fast
1765 // cause an issue if network connection is slow
1766 > db.persons.insertOne({name: 'Aliya', age: 35},{writeConcern: {w: 1, j:
true,wtimeout: 1}})
1767 {
1768   "acknowledged" : true,
1769   "insertedId" : ObjectId("5f154309e3242ab6a2f87b57")
1770 }
1771
1772
1773 -----Atomicity-----
1774
1775 1. Operation (e.g. insertOne()) --> Error --> Rolled Back(i.e NOTHING is saved)
1776 2. Operation (e.g. insertOne()) --> Success --> Saved as Whole
1777
1778 ## The Atomicity guarantees that an atomic transaction which means the transaction
either succeeds as a whole or it fails as a whole.
1779
1780 ## I it fails during the write, everything is rolled back for this document that
are inserted.
1781
1782 ## Its on a per document level, that means the top level document, it includes all
embedded documents, all arrays so that is all included.
1783
1784 ## MongoDB CRUD operation are Atomic on the Document Level(including Embedded
Documents).
1785
1786
-----
---
1787
1788 ## If you use insert many with multiple documents being inserted, then you don't
get this.
1789
1790 ## If you have multiple documents in one operation, like insert many, the only each
document on its own is guaranteed to either fail or succeed but not insert many.
1791
1792 ## Here does not roll back if any occurs create in one document.
1793
1794 -----Importing Data-----
1795
1796 mongoimport tv-shows.json -d movieData -c movies --jsonArray --drop
1797

```

```

1798 --jsonArray -> to make the mongo import command aware of this.
1799 --drop -> collection should already exist, it will dropped and then re-added
        otherwise it we'll append the data to the existing collection and that might also
        be what you want.

```

```

1800
1801
1802
1803
1804

```

```

1805 06-crud-operations-advanced
1806 7-read(part-1)
1807

```

```

1808
1809 -----READ-----
1810

```

- 1811 1. Methods, Filters & Operations
- 1812 2. Query Selectors (READ)
- 1813 3. Projection Operators (READ)

1814 Sample Example :

```

1815 1. db.myCollection.find({age: 30})
1816 here {age: 30 } --> Filter . age -> Field, 32 --> Value
1817

```

```

1818 2. db.myCollection.find({age: { $gt: 30}})
1819 {age: { $gt: 30}} --> Filter(Range) &gt; --> Operator
1820

```

```

1821 ----- Operator-----
1822

```

1823	Read	Update		
1824	Query & Projection	Update	Query Modifiers	Aggregation
1825	Query Selectors----->Fields----->Change Query----->Pipeline			
1826	Stages			

1827	Projection Operators----->Arrays----->	Behaviors	----->Pipeline Operators
1828			
1829			
1830		This is Deprecated now	

```

1831 -----How Operators Impact Our Data-----
1832

```

1833	Type	Purpose	Changes Data?	Example
1834	Query Operator----->Locate Data----->	blocked----->	\$eq	
1835	Projection Operator----->Modify data presentation----->	blocked----->	\$	
1836	Update Operator-----> Modify + add additional----->	not blocked----->	&inc	
1837		data		

```

1838 -----Query Selectors types-----
1839

```

- 1840 1. Comparison 2. Logical 3. Element 4. Evaluation 5. Array 6. Comments 7.
- 1841 Geaspatial(special)

```

1842 ----- Projections Operators-----
1843

```

- 1844 1. \$ 2. \$elemMatch 3. \$meta 4. \$slice

```

1845
1846 > use movieData
1847 switched to db movieData
1848 > cls
1849

```

```

1850 > db.movies.findOne()
1851 {
1852   "_id" : ObjectId("5f15a22a9bfbc37d06f66616"),
1853

```

```

1861     "id" : 1,
1862     "url" : "http://www.tvmaze.com/shows/1/under-the-dome",
1863     "name" : "Under the Dome",
1864     "type" : "Scripted",
1865     "language" : "English",
1866     "genres" : [
1867         "Drama",
1868         "Science-Fiction",
1869         "Thriller"
1870     ],
1871     "status" : "Ended",
1872     "runtime" : 60,
1873     "premiered" : "2013-06-24",
1874     "officialSite" : "http://www.cbs.com/shows/under-the-dome/",
1875     "schedule" : {
1876         "time" : "22:00",
1877         "days" : [
1878             "Thursday"
1879         ]
1880     },
1881     "rating" : {
1882         "average" : 6.5
1883     },
1884     "weight" : 91,
1885     "network" : {
1886         "id" : 2,
1887         "name" : "CBS",
1888         "country" : {
1889             "name" : "United States",
1890             "code" : "US",
1891             "timezone" : "America/New_York"
1892         }
1893     },
1894     "webChannel" : null,
1895     "externals" : {
1896         "tvrage" : 25988,
1897         "thetvdb" : 264492,
1898         "imdb" : "tt1553656"
1899     },
1900     "image" : {
1901         "medium" : "http://static.tvmaze.com/uploads/images/medium_portrait/0/1.jpg",
1902         "original" :
1903             "http://static.tvmaze.com/uploads/images/original_untouched/0/1.jpg"
1904     },
1905     "summary" : "<p><b>Under the Dome</b> is the story of a small town that is
1906 suddenly and inexplicably sealed off from the rest of the world by an enormous
1907 transparent dome. The town's inhabitants must deal with surviving the
1908 post-apocalyptic conditions while searching for answers about the dome, where
1909 it came from and if and when it will go away.</p>",
1910     "updated" : 1529612668,
1911     "_links" : {
1912         "self" : {
1913             "href" : "http://api.tvmaze.com/shows/1"
1914         },
1915         "previousepisode" : {
1916             "href" : "http://api.tvmaze.com/episodes/185054"
1917         }
1918     }
1919 }
1920
1921 ----- 1.Comparison-----
1922
1923 $ne, $eq, $lt, $lte, $gt, $gte, $in, $nin
1924
1925 > db.movies.find({runtime: 60}).pretty()
1926 > db.movies.findOne({runtime: 60})
1927
1928 // exactly the same
1929 >db.movies.findOne({runtime: {$eq: 60}})

```

```

1925
1926 > db.movies.find({runtime: {$ne: 60}}).pretty()
1927 > db.movies.find({runtime: {$lt: 40}}).pretty()
1928 > db.movies.find({runtime: {$lte: 40}}).pretty()
1929 > db.movies.find({runtime: {$gt: 40}}).pretty()
1930 > db.movies.find({runtime: {$gte: 40}}).pretty()
1931
1932 // query into imbedded documents
1933 N.B : In imbedded documents have to use must quotes
1934
1935 > db.movies.find({"rating.average": {$gt: 7}}).pretty()
1936
1937 // query into imbedded array
1938
1939 > db.movies.find({genres:"Drama"}).pretty()
1940 // to exact query
1941 > db.movies.find({genres:["Drama"]}).pretty()
1942
1943 // it will find us all documents that have a runtime of 30 or 42 but not 60
1944
1945 [30,42] --> this is set of values not a range
1946 > db.movies.find({runtime: {$in:[30,42]}}).pretty()
1947 > db.movies.find({runtime: {$nin:[30,42]}}).pretty()
1948
1949
1950 ----- 2.Logical-----
1951
1952 $or, $and, $not, $nor
1953
1954 // multiple query
1955
1956 > db.movies.find({$or: [{"rating.average": {$lt: 5}}, {"rating.average": {$gt:
1957 9.3}]}]).count()
1958 > db.movies.find({$or: [{"rating.average": {$lt: 5}}, {"rating.average": {$gt:
1959 9.3}]}]).pretty()
1960
1961 > db.movies.find({$nor: [{"rating.average": {$lt: 5}}, {"rating.average": {$gt:
1962 9.3}]}]).count()
1963
1964 // this is the older command
1965 > db.movies.find({$and: [{"rating.average": {$gt: 9}}, {"genres : "Drama"}]}).count()
1966 > db.movies.find({$and: [{"rating.average": {$gt: 9}}, {"genres : "Drama"}]}).pretty()
1967
1968 // latest command (using only document)
1969 > db.movies.find({"rating.average": {$gt: 9}, genres : "Drama"}).count()
1970
1971 // this basically not work, does not give right value
1972 // same object is not permitted int this way
1973 // here issue is create by same json key genres, this keys value replace the first
1974 one when execute second
1975
1976 > db.movies.find({genres : "Drama", genres: 'Horror'}).count()
1977
1978 // this also have same result
1979 > db.movies.find({genres: 'Horror'}).count()
1980
1981 // we have to use and in the same field
1982 // in this issue we have to use $and must
1983 > db.movies.find({$and: [{genres : "Drama"}, {genres: 'Horror'}]}).count()
1984 > db.movies.find({runtime: {$not :{$eq: 60}}}).count()
1985
1986 // this is also equal to the $ne
1987 > db.movies.find({runtime: {$ne: 60}}).count()
1988
1989 ----- 3.Element-----
1990 $exists $type

```

```

1990 // exists
1991
1992 > db.users.insertMany([{name: 'Max', hobbies: [{title: 'Sports', frequency:
3},{title: 'Cooking', frequency: 6}], phone: 0123495334},{name: 'Manuel', hobbies:
[{title: 'Cooking', frequency: 5},{title: 'Cars', frequency: 6}], phone:
'043453495334', age: 30}])
1993 {
1994   "acknowledged" : true,
1995   "insertedIds" : [
1996     ObjectId("5f172a343a76a40cd42b836a"),
1997     ObjectId("5f172a343a76a40cd42b836b")
1998   ]
1999 }
2000 > db.users.find().pretty()
2001 {
2002   "_id" : ObjectId("5f172a343a76a40cd42b836a"),
2003   "name" : "Max",
2004   "hobbies" : [
2005     {
2006       "title" : "Sports",
2007       "frequency" : 3
2008     },
2009     {
2010       "title" : "Cooking",
2011       "frequency" : 6
2012     }
2013   ],
2014   "phone" : 123495334
2015 }
2016 {
2017   "_id" : ObjectId("5f172a343a76a40cd42b836b"),
2018   "name" : "Manuel",
2019   "hobbies" : [
2020     {
2021       "title" : "Cooking",
2022       "frequency" : 5
2023     },
2024     {
2025       "title" : "Cars",
2026       "frequency" : 6
2027     }
2028   ],
2029   "phone" : "043453495334",
2030   "age" : 30
2031 }
2032
2033 // here checking an element exists or not
2034 > db.users.find({age: {$exists: true}}).pretty()
2035 {
2036   "_id" : ObjectId("5f172a343a76a40cd42b836b"),
2037   "name" : "Manuel",
2038   "hobbies" : [
2039     {
2040       "title" : "Cooking",
2041       "frequency" : 5
2042     },
2043     {
2044       "title" : "Cars",
2045       "frequency" : 6
2046     }
2047   ],
2048   "phone" : "043453495334",
2049   "age" : 30
2050 }
2051
2052 // can also check multiple logic
2053 > db.users.find({age: {$exists: true, $gt: 30}}).pretty()
2054 > db.users.find({age: {$exists: true, $gte: 30}}).pretty()
2055

```



```

2056 // if element value is null then it also be exists
2057 > db.users.insertMany([
2058   {name: 'Anna', hobbies: [{title: 'Sports', frequency:
2059     2},{title: 'Yoga', frequency: 3}], phone: 01234953345, age: null}])
2060 {
2061   "acknowledged" : true,
2062   "insertedIds" : [
2063     ObjectId("5f172c593a76a40cd42b836c")
2064   ]
2065 }
2066 > db.users.find({age: {$exists: true}}).pretty()
2067
2068 // but if we check with exists value is false and if an element value has null but
2069 exist then it also does not show
2070 > db.users.find({age: {$exists: false}}).pretty()
2071
2072 // checking exit and value not equal null
2073 > db.users.find({age: {$exists: true, $ne: null}}).pretty()
2074
2075 // type
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120

```

Type	Number	Alias	Notes
Double	1	"double"	
String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary data	5	"binData"	
Undefined	6	"undefined"	Deprecated.
ObjectId	7	"objectId"	
Boolean	8	"bool"	
Date	9	"date"	
Null	10	"null"	
Regular Expression	11	"regex"	
DBPointer	12	"dbPointer"	Deprecated.
JavaScript	13	"javascript"	
Symbol	14	"symbol"	Deprecated.
JavaScript (with scope)	15	"javascriptWithScope"	
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit integer	18	"long"	
Decimal128	19	"decimal"	New in version 3.4.
Min key	-1	"minKey"	
Max key	127	"maxKey"	

```

2112 // checking with alias
2113 > db.users.find({phone: {$type: 'number'}}).pretty()
2114
2115 // as shell is based on JavaScript number and double would be the same answer
2116 // In database number is stored into floating point number as double
2117 // JS drivers only knows it always double
2118
2119 > db.users.find({phone: {$type: 'double'}}).pretty()
2120
2121 // also can add multiple type
2122 > db.users.find({phone: {$type: ['double', 'string']}}).pretty()

```

---

06-crud-operations-advanced

7-read(part-2)

---

-----4.Evaluation Operators-----

\$expr, \$regex, \$text, \$where

```

2121 // $regex allows us search for text
2122
2123 // return the document that found the word
2124 // it is not best way to find text in this way
2125 > db.movies.find({summary: {$regex: /musical/}}).pretty()
2126
2127
2128 $expr --> compare two fields inside one document then return that fields
2129
2130 > use financialDatalet retrieve_code = request.params.id;
2131 switched to db financialData
2132 > db.sales.insertMany([{volume: 100, target: 120},{volume: 89, target: 80},{volume:
2133 200, target: 177}])
2134
2135 > db.sales.find().pretty()
2136 {
2137   "_id" : ObjectId("5f17491c3a76a40cd42b836d"),
2138   "volume" : 100,
2139   "target" : 120
2140 }
2141 {
2142   "_id" : ObjectId("5f17491c3a76a40cd42b836e"),
2143   "volume" : 89,
2144   "target" : 80
2145 }
2146 {
2147   "_id" : ObjectId("5f17491c3a76a40cd42b836f"),
2148   "volume" : 200,
2149   "target" : 177
2150 }
2151
2152 ## we want to find all entries, all items in this collection where the volume is
2153 above the target
2154
2155 // we have to use double quotes in to query, have to pass reference fields name
2156 // have to use dollar sign before fields also
2157
2158 // this will not work
2159 > db.sales.find({$expr: {$gt: ['volume', 'target']}}).pretty()
2160
2161 // this work successfully
2162 > db.sales.find({$expr: {$gt: ['$volume', '$target']}}).pretty()
2163
2164 {
2165   "_id" : ObjectId("5f17491c3a76a40cd42b836e"),
2166   "volume" : 89,
2167   "target" : 80
2168 }
2169 {
2170   "_id" : ObjectId("5f17491c3a76a40cd42b836f"),
2171   "volume" : 200,
2172   "target" : 177
2173 }
2174
2175 ## we do not want to compare whether volume is greater than target and also (want
2176 to find if volume is above 190 and the difference to target at least 10)
2177
2178 // this is more complex query
2179 $cond --> conditional because we are in document
2180
2181 > db.sales.find({$expr: {$gt: [{ $cond: {if: {$gte: ['$volume', 190]}, then:
2182 {$subtract: ['$volume', 10]}, else: '$volume'}}, '$target']}}).pretty()
2183
2184 {
2185   "_id" : ObjectId("5f17491c3a76a40cd42b836e"),
2186   "volume" : 89,
2187   "target" : 80
2188 }
2189 {
2190   "_id" : ObjectId("5f17491c3a76a40cd42b836f"),

```

```

2186         "volume" : 200,
2187         "target" : 177
2188     }
2189
2190     // if i increase the subtracted value logically then result might be changed
2191
2192     > db.sales.find({$expr: {$gt: [{ $cond: {if: {$gte: ['$volume', 190]}, then:
2193     {$subtract: ['$volume', 30]}, else: '$volume'}}], '$target'}}}).pretty()
2194     {
2195         "_id" : ObjectId("5f17491c3a76a40cd42b836e"),
2196         "volume" : 89,
2197         "target" : 80
2198     }
2199
2200     -----5.Array-----
2201
2202     $elemMatch, $size , $all
2203
2204     > use user
2205     switched to db user
2206
2207     // find all hobbies that are sports
2208     // this won't work
2209     > db.users.find({hobbies: 'Sports'}).pretty()
2210
2211     // also find nothing when using nested document
2212     > db.users.find({hobbies: {title: 'Sports'}}).pretty()
2213
2214     // this also can not a perfect value
2215     > db.users.find({hobbies: {title: 'Sports', frequency: 2}}).pretty()
2216
2217     // act an embedded document
2218     // this is path embedded approach not only on a directly embedded documents
2219     // this is similar to multiple embedded documents query
2220
2221     > db.users.find({'hobbies.title': 'Sports'}).pretty()
2222     {
2223         "_id" : ObjectId("5f172a343a76a40cd42b836a"),
2224         "name" : "Max",
2225         "hobbies" : [
2226             {
2227                 "title" : "Sports",
2228                 "frequency" : 3
2229             },
2230             {
2231                 "title" : "Cooking",
2232                 "frequency" : 6
2233             }
2234         ],
2235         "phone" : 123495334
2236     }
2237     {
2238         "_id" : ObjectId("5f172c593a76a40cd42b836c"),
2239         "name" : "Anna",
2240         "hobbies" : [
2241             {
2242                 "title" : "Sports",
2243                 "frequency" : 2
2244             },
2245             {
2246                 "title" : "Yoga",
2247                 "frequency" : 3
2248             }
2249         ],
2250         "phone" : 1234953345,
2251         "age" : null
2252     }
2253

```

```

2254 ## want to find all users who have exactly 3 hobbies
2255 > db.users.insertOne({name: 'Chris', hobbies: ['Sports', 'Cooking', 'Hiking']})
2256 > db.users.find({'hobbies': {$size: 3}}).pretty()
2257
2258 ## if want to query like hobbies greater than 3 or smaller. It does not support
2259 MongoDB
2260 > use boxOffice
2261 > db.movies.find().pretty()
2262
2263 ## want to find movies that have a genre of exactly thriller and action
2264 // this won't work perfectly, here basically work with index ordering and also exact
2265 // we are not concern about ordering
2266 > db.movies.find({'genre': ['action', 'thriller']}).pretty()
2267
2268 // $all basically find if array have all fields like 'action', 'thriller'
2269 > db.movies.find({'genre': {$all: ['action', 'thriller']}}).pretty()
2270
2271 ## want to find all users who have a hobby of sports and the frequency should be
2272 grater or equal to 2
2273 > db.users.find({$and: [{'hobbies.title': 'Sports'}, {'hobbies.frequency':
2274 {$gte: 2}]}]).pretty()
2275
2276 // if we change the query replace 2 with 3. does not work properly
2277 {
2278   "_id" : ObjectId("5f172a343a76a40cd42b836a"),
2279   "name" : "Max",
2280   "hobbies" : [
2281     {
2282       "title" : "Sports",
2283       "frequency" : 3
2284     },
2285     {
2286       "title" : "Cooking",
2287       "frequency" : 6
2288     }
2289   ],
2290   "phone" : 123495334
2291 }
2292 {
2293   "_id" : ObjectId("5f172c593a76a40cd42b836c"),
2294   "name" : "Anna",
2295   "hobbies" : [
2296     {
2297       "title" : "Sports",
2298       "frequency" : 2
2299     },
2300     {
2301       "title" : "Yoga",
2302       "frequency" : 3
2303     }
2304   ],
2305   "phone" : 1234953345,
2306   "age" : null
2307 }
2308 // this work with different embedded document but we do not want that
2309 // does not give the exact result
2310 > db.users.find({$and: [{'hobbies.title': 'Sports'}, {'hobbies.frequency':
2311 {$gte: 3}]}]).pretty()
2312 {
2313   "_id" : ObjectId("5f172a343a76a40cd42b836a"),
2314   "name" : "Max",
2315   "hobbies" : [
2316     {
2317       "title" : "Sports",
2318       "frequency" : 3

```

```

2319         {
2320             "title" : "Cooking",
2321             "frequency" : 6
2322         }
2323     ],
2324     "phone" : 123495334
2325 }
2326 {
2327     "_id" : ObjectId("5f172c593a76a40cd42b836c"),
2328     "name" : "Anna",
2329     "hobbies" : [
2330         {
2331             "title" : "Sports",
2332             "frequency" : 2
2333         },
2334         {
2335             "title" : "Yoga",
2336             "frequency" : 3
2337         }
2338     ],
2339     "phone" : 1234953345,
2340     "age" : null
2341 }
2342 > db.users.find({$and: [{'hobbies.title': 'Sports'},{'hobbies.frequency':
{$gte:3}]}])).pretty().count()
2343
2344 // we want to ensure that query have to perform into same document/element
2345 // work properly
2346 // perform query into single document
2347
2348 > db.users.find({hobbies: {$elemMatch:{title: 'Sports', frequency: {$gte:
3}}}}).pretty().pretty()
2349 {
2350     "_id" : ObjectId("5f172a343a76a40cd42b836a"),
2351     "name" : "Max",
2352     "hobbies" : [
2353         {
2354             "title" : "Sports",
2355             "frequency" : 3
2356         },
2357         {
2358             "title" : "Cooking",
2359             "frequency" : 6
2360         }
2361     ],
2362     "phone" : 123495334
2363 }
2364
2365 -----Understanding Cursors-----
2366
2367 // when we find() it basically getting the all data like 100 millions
2368 // it can reduce if we include query
2369
2370 // here cursors basically a pointer
2371 // cursor request batch to the server every time to get the data
2372
2373 // If have a query that meets 1000 documents, but let's say you have a website
2374 // where you only display 10 items, let's say 10 products you fetched at a time
2375 // anyways, then there is no need to load all thousand results that matched your query
2376 // right at the start. Instead you would only fetch the first 10, display them on the
2377 // screen and then go ahead and fetch the next 10, when the user navigated to the next
2378 // page or anything like that. This is the idea behind a cursor.
2379
2380 > use movieData
2381 > db.movies.find().count()
2382 240
2383
2384 // basically it returns 20 elements

```

```

2381 > db.movies.find().pretty()
2382
2383 // type it for more
2384 > it
2385
2386 // get exactly one document, because next() gives the next document
2387 > db.movies.find().next()
2388
2389 // we can use JavaScript syntax in mongoShell
2390 > const dataCursor = db.movies.find()
2391 > dataCursor.next()
2392 > dataCursor.next()
2393
2394 // printjson() is a mongoShell function that helps to print something into shell
2395
2396 // fetched all documents
2397 > dataCursor.forEach(document => {printjson(document)})
2398
2399 // check have any next value
2400 > dataCursor.hasNext()
2401
2402 // fetching data with sort()
2403 // one means ascending
2404 > db.movies.find().sort({'rating.average': 1}).pretty()
2405
2406 // minus one mean descending
2407 > db.movies.find().sort({'rating.average': -1}).pretty()
2408
2409 // sort with multiple query
2410 // here first sort the 'rating.average' and if 'rating.average' have same value
2411 // into particular indexes and then runtime execute with descending if may exist
2412 > db.movies.find().sort({'rating.average': 1, runtime: -1}).pretty()
2413
2414 // next() also exist with sort()
2415 > db.movies.find().sort({'rating.average': 1, runtime: -1}).next()
2416
2417 // we can skip certain amount of elements
2418 // it effective in pagination
2419 // when we work with pagination we can skip the previous 10 elements
2420 > db.movies.find().sort({'rating.average': 1, runtime: -1}).skip(10).pretty()
2421
2422 // skip() basically limit the amount of elements the cursor should retrieve at a time
2423 // can still have include limit
2424
2425 // limit return the exact number of element
2426 > db.movies.find().sort({'rating.average': 1, runtime:
2427 -1}).skip(100).limit(10).pretty()
2428
2429 // here order does not matter.
2430 // Order check from right such(previousexample) : sort()->skip()->limit()
2431 // what method we write first, it execute first
2432
2433 -----Using Projection to Share our Results-----
2434
2435 ## want to retrieve elements with specific fields
2436 // we have no to check the other fields. they are executed by default
2437
2438 // here ID always include
2439 > db.movies.find({}, {name: 1, genres: 1, runtime: 1, rating: 1}).pretty()
2440
2441 // to exclude the ID
2442 > db.movies.find({}, {name: 1, genres: 1, runtime: 1, rating: 1, _id: 0}).pretty()
2443
2444 // can also use embedded document with path notation
2445 > db.movies.find({}, {name: 1, genres: 1, runtime: 1, 'rating.average': 1, _id:
2446 0}).pretty()

```

```

2447 'schedule.time': 1, _id: 0)).pretty()
2448
2449 // can also add logic
2450 > db.movies.find({'rating.average': {$gt: 8}}, {name: 1, genres: 1, runtime: 1,
2451 'rating.average': 1, 'schedule.time': 1, _id: 0}).pretty()
2452
2453 -----Projection in Arrays-----
2454
2455 // simply array query
2456 > db.movies.find({genres: 'Drama'}).pretty()
2457
2458 // return the array projection of related query
2459 > db.movies.find({genres: 'Drama'}, {'genres.$': 1}).pretty()
2460
2461 // it does not work properly
2462 > db.movies.find({genres: {$all: ['Drama', 'Horror']}}, {'genres.$': 1}).pretty()
2463
2464 // this projection is element wise and exact query
2465 // {$elemMatch: {$eq: 'Horror'}} --> this thing decide which item is displayed or not
2466 > db.movies.find({genres: 'Drama'}, {genres: {$elemMatch: {$eq: 'Horror'}}}).pretty()
2467
2468 // can also check with other element
2469 > db.movies.find({'rating.average': {$gt: 9}}, {genres: {$elemMatch: {$eq:
2470 'Horror'}}}).pretty()
2471
2472 -----Projection Slice-----
2473
2474 // slicing array that i want
2475 // can add any number
2476 // $slice: 2 --> how many array elements we want to show from first
2477 > db.movies.find({'rating.average': {$gt: 9}}, {genres: {$slice: 2}, name:
2478 1}).pretty()
2479
2480 // can also be execute with array from
2481 // 1--> What lengths of elements we want to skip (index - start from 1)
2482 // 2--> How many element we want to show
2483 > db.movies.find({'rating.average': {$gt: 9}}, {genres: {$slice: [1, 2]}, name:
2484 1}).pretty()
2485
2486 // checking
2487 > db.movies.find({_id: ObjectId("5f15a22a9bfb37d06f66662")}, {genres: 1}).pretty()
2488
2489 {
2490   "_id" : ObjectId("5f15a22a9bfb37d06f66662"),
2491   "genres" : [
2492     "Drama",
2493     "Adventure",
2494     "Fantasy"
2495   ]
2496 }
2497
2498 > db.movies.find({'rating.average': {$gt: 9}}, {genres: {$slice: [2, 2]}, name:
2499 1}).pretty()
2500
2501 {
2502   "_id" : ObjectId("5f15a22a9bfb37d06f66662d"),
2503   "name" : "Berserk",
2504   "genres" : [
2505     "Horror"
2506   ]
2507 }
2508
2509 {
2510   "_id" : ObjectId("5f15a22a9bfb37d06f66662"),
2511   "name" : "Game of Thrones",
2512   "genres" : [
2513     "Fantasy"
2514   ]
2515 }

```

```

2510 {
2511     "_id" : ObjectId("5f15a22a9bfbc37d06f666b7"),
2512     "name" : "Breaking Bad",
2513     "genres" : [
2514         "Thriller"
2515     ]
2516 }
2517 {
2518     "_id" : ObjectId("5f15a22a9bfbc37d06f666c0"),
2519     "name" : "The Wire",
2520     "genres" : [ ]
2521 }
2522 {
2523     "_id" : ObjectId("5f15a22a9bfbc37d06f666c1"),
2524     "name" : "Firefly",
2525     "genres" : [
2526         "Western"
2527     ]
2528 }
2529 {
2530     "_id" : ObjectId("5f15a22a9bfbc37d06f666d8"),
2531     "name" : "Stargate SG-1",
2532     "genres" : [
2533         "Science-Fiction"
2534     ]
2535 }
2536 {
2537     "_id" : ObjectId("5f15a22a9bfbc37d06f666e2"),
2538     "name" : "Rick and Morty",
2539     "genres" : [
2540         "Science-Fiction"
2541     ]
2542 }
2543
2544
2545
2546 -----
2547 06-crud-operations-advanced
2548 8-update
2549 -----
2550
2551 -----Update-----
2552
2553 1. Document Updating Operator 2. Updating Fields 3. Updating Arrays
2554
2555 -----1. Document Updating Operator-----
2556
2557 $min, $max, $mul, $inc, $set, $unset
2558
2559 // set basically changed or added new document
2560
2561 > use user
2562 switched to db user
2563
2564 > db.users.find().pretty()
2565
2566 > db.users.updateOne({_id: ObjectId("5f17c3d47122dce4fa46fb4a")}, {$set:
{'hobbies':[{'title: 'Sports', frequency: 5},{title: 'Cooking', frequency: 3}, {title:
'Hiking', frequency: 1}]}})
2567
2568 // $set basically add a new value or update existing value
2569 // update the users whose hobby is Sports
2570 > db.users.updateMany({'hobbies.title': 'Sports'}, {$set: {isSporty: true}})
2571
2572 // adding multiple elements using $set
2573 > db.users.updateOne({_id: ObjectId("5f17c3d47122dce4fa46fb4a")}, {$set: {age: 40,
phone: 082344289399}})

```



```

2574 // can manually increment or decrement any number document
2575 > db.users.updateOne({name: "Manuel"}, {$inc: {age: 2}})
2576 > db.users.updateOne({name: "Manuel"}, {$inc: {age: -2}})
2577
2578 // also check with multiple query
2579 > db.users.updateOne({name: "Manuel"}, {$inc: {age: -2}, $set:{isSporty: false}})
2580
2581 // two operations is not allowed into same fields.
2582 > db.users.updateOne({name: "Manuel"}, {$inc: {age: -2}, $set:{age: 30}})
2583
2584 // update the age value with min value
2585 > db.users.updateOne({name: "Chris"}, {$min: {age: 35}})
2586 > db.users.updateOne({name: "Chris"}, {$max: {age: 39}})
2587
2588 // multiply age by a number specify 10 %
2589 > db.users.updateOne({name: "Chris"}, {$mul: {age: 1.1}})
2590
2591 -----2. Updating Fields-----
2592
2593 $upsert
2594
2595 // Getting Rid of Fields
2596
2597 ## want to drop all value on all persons who are sporty
2598 > db.users.updateMany({isSporty: true}, {$set: {phone: null}})
2599
2600 // to get rid of fields
2601 > db.users.updateMany({isSporty: true}, {$unset: {phone: null}})
2602
2603 > db.users.updateMany({isSporty: true}, {$unset: {phone: ''}})
2604
2605 // Renaming Fields
2606
2607 $rename, $set, upsert
2608
2609 > db.users.updateMany({}, {$rename: {age: 'totalAge'}})
2610 { "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 3 }
2611
2612 ## want to update some fields but does not know if its exist or not.
2613 // if it is exist then override the document
2614 // if it does not exist then create new document
2615
2616 // its normal insert and update
2617 > db.users.updateOne({name: 'Maria'}, {$set: {age: 29, hobbies: [{title: 'Good
2618 food', frequency: 3}], isSporty: true}})
2619 { "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
2620
2621 > db.users.updateOne({name: 'Maria'}, {$set: {age: 29, hobbies: [{title: 'Good
2622 food', frequency: 3}], isSporty: true}}, {upsert: false})
2623
2624 // this works perfectly
2625 // work with filter
2626 > db.users.updateOne({name: 'Maria'}, {$set: {age: 29, hobbies: [{title: 'Good
2627 food', frequency: 3}], isSporty: true}}, {upsert: true})
2628
2629 -----3. Updating Arrays-----
2630
2631 $., $[], $push, $pop, $pull, $each, $addToSet
2632
2633 ## want to find a certain amount of persons and persons based on the hobbies array
2634
2635 // checking query into array same elements but not working perfectly
2636 > db.users.find({$and: [{hobbies.title: 'Sports'}, {hobbies.frequency:
2637 {$gte: 3}]}}).pretty()
2638
2639 // this is the exact query
2640 > db.users.find({hobbies: {$elemMatch: {title: 'Sports', frequency:
2641 {$gte: 3}}}}).pretty()

```

```

2638
2639 // updating hole matched array elements
2640 // here .$ -> this will automatically refer to the element in our filter as i want
    to update the element in hobbies which matched the condition
2641 // $--> dollar sign is a place holder here
2642
2643 // adding new field
2644 > db.users.updateMany({hobbies: {$elemMatch: {title: 'Sports',frequency:
    {$gte:3}}}}, {$set : {"hobbies.$.highFrequency": true}})
2645
2646 // updating All Array Elements
2647 > db.users.find({'hobbies.frequency': {$gt: 2}}).pretty()
2648 > db.users.find({'hobbies.frequency': {$gt: 2}}).count()
2649
2650 // updating the elements
2651 // but this won't work properly
2652 > db.users.updateMany({'hobbies.frequency': {$gt: 2}}, {$set:
    {'hobbies.$.goodFrequency': true}})
2653
2654 ## lets say if totalAge is greater than 30 than we want to update the every array
    elements
2655 // do not override
2656 // .${} --> update all array elements and for each element because we have embedded
    document.
2657 > db.users.updateMany({totalAge: {$gt: 30}}, {$inc: {'hobbies.${}.frequency': -1}})
2658
2659 ## want to find all hobbies with a frequency greater than 2
2660 > db.users.find({'hobbies.frequency': {$gt: 2}}).pretty()
2661
2662 // el --> is a identifier for every items of array into documents
2663 // {'hobbies.frequency': {$gt: 2}} --> this filter identify documents
2664 // {'el.frequency': {$gt: 2}} --> this filter which identify array elements
2665 // these two are not equal
2666
2667 > db.users.updateMany({'hobbies.frequency': {$gt: 2}}, {$set:
    {'hobbies.${el}.goodFrequency': true}}, {arrayFilters: [{'el.frequency': {$gt: 2}}]})
2668 { "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 2 }
2669
2670 // Adding Elements to Arrays
2671 // taking also old array
2672 > db.users.updateOne({name: 'Maria'}, {$push: {hobbies: {title: 'Sports',
    frequency: 2}}})
2673
2674 // $push also used with more than one document
2675 // also use $sort, $sort is related with every $each
2676 > db.users.updateOne({name: 'Maria'}, {$push: {hobbies: {$each: [{title: 'Good
    Wine', frequency: 1}, {title: 'Good Wine', frequency: 2}], $sort: {frequency: -1}}}})
2677
2678 // Removing Elements from array
2679 // $pull describe an object that what we want to pull
2680 > db.users.updateOne({name: 'Maria'}, {$pull: {hobbies: {title: 'Hiking'}}})
2681
2682 // Remove the last element of an array
2683 > db.users.updateOne({name: 'Chris'}, {$pop: {hobbies: 1}})
2684
2685 // Remove the first element of an array
2686 > db.users.updateOne({name: 'Chris'}, {$pop: {hobbies: -1}})
2687
2688 // Understanding $addToSet
2689 // $addToSet adds unique value only
2690 // its basically add new element. But if the element already exist(have to exact
    same) it does not update.
2691 > db.users.updateOne({name: 'Maria'}, {$addToSet: {hobbies: {title: 'Hiking',
    frequency: 2}}})
2692
2693
2694
2695 -----
    -----

```

```

2696 06-crud-operations-advanced
2697 9-delete
2698 -----
2699 -----
2700 -----Delete-----
2701
2702 > use user
2703 switched to db user
2704 > db.users.deleteOne({name: 'Chris'})
2705
2706 // delete with matched query
2707 > db.users.deleteOne({totalAge: {$gt: 30}, isSporty: true})
2708
2709 > db.users.deleteOne({totalAge: {$exists: false}, isSporty: true})
2710
2711 // delete many
2712 > db.users.deleteMany({totalAge: {$gt: 30}, isSporty: true})
2713
2714 // deleting all entries in a collection
2715 // {} --> this is simply is a filter that matches every document in the collection.
2716 > db.users.deleteMany({})
2717
2718 // to delete the entire collection
2719 > db.users.drop()
2720
2721 // to delete the entire dataBase
2722 > db.dropDataBase()
2723
2724
2725
2726
2727 -----
2728 -----
2729 10-working-with-indexes
2730 credit-rating
2731 -----
2732 -----
2733
2734 conn = new Mongo();
2735 db = conn.getDB("credit");
2736
2737 for (let i = 0; i < 1000000; i++) {
2738   db.ratings.insertOne({
2739     "person_id": i + 1,
2740     "score": Math.random() * 100,
2741     "age": Math.floor(Math.random() * 70) + 18
2742   })
2743 }
2744
2745 -----
2746 -----
2747 10-working-with-indexes
2748 -----
2749 -----
2750 -----Index and others-----
2751
2752 // different Types of Indexes
2753 // Using and Optimizing Indexes
2754 // indexes are order list of values
2755 // Its point related index just like a pointer indexes
2756 // Indexes are updated with every insert
2757
2758 ----- What ant Why -----
2759 1. Indexes allow to retrieve data more efficiently (if used correctly) because
2760 queries only have to look at a subset of all documents.

```

```

2759
2760 2. Can use single-field, compound, multi-key(array) and text indexes.
2761
2762 3. Indexes don't come for free, they will slow down writes.
2763 ----- Queries & Sorting -----
2764 4. Indexes can be used for both queries and efficient sorting.
2765
2766 5. Compound indexes can be used as a whole or in a 'left-to-right' (prefix) manner
    (e.g only consider the 'name' of the 'name-age' compound index)
2767
2768
2769 // Adding a Single Field Index
2770 > use contactData
2771 > db.contacts.find({'dob.age': {$gt: 60}}).pretty()
2772
2773 //analyze database with explain() method
2774 > db.contacts.explain().find({'dob.age': {$gt: 60}})
2775
2776 // here also have a different types of plan --> 1. winningPlan 2. rejectedPlans
2777 // getting the detailed query
2778 > db.contacts.explain("executionStats").find({'dob.age': {$gt: 60}})
2779
2780 // creating index, index is defined as a document
2781 // here one basically means i want to sort the data with ascending order
2782 > db.contacts.createIndex({'dob.age': 1})
2783 {
2784   "createdCollectionAutomatically" : false,
2785   "numIndexesBefore" : 1,
2786   "numIndexesAfter" : 2,
2787   "ok" : 1
2788 }
2789
2790 // if change the query then it works like before
2791 // Without index every different query ha different values
2792 > db.contacts.explain("executionStats").find({'dob.age': {$gt: 20}})
2793
2794 // dropping the index
2795 > db.contacts.dropIndex({'dob.age': 1})
2796 { "nIndexesWas" : 2, "ok" : 1 }
2797
2798 // Understanding Index Restrictions
2799 > db.contacts.explain("executionStats").find({'dob.age': {$gt: 20}})
2800
2801 // if have to retrieve large number of documents nearly 70 to 80% then index can
    effect the query be slower,cause for using query we have to add an extra step.
2802
2803 // for retrieving 20-30% or lower can using index, query be faster
2804 // creating a compound index with text
2805 > db.contacts.createIndex({gender: 1})
2806 // get explain after creating index
2807 > db.contacts.explain("executionStats").find({gender: "male"})
2808
2809 ## want to find all persons who are older than 30 and male or older than 40 and male
2810 // when using multiple fields for query in index, basically one combined index is
    created from multiple fields.
2811 // here created one combined index from two fields
2812 // every time have to drop if uses the same filed
2813 > db.contacts.dropIndex({'gender': 1})
2814 > db.contacts.createIndex({'dob.age': 1,'gender': 1})
2815
2816 // getting info from query
2817 > db.contacts.explain().find({'dob.age': 35,'gender': 'male'})
2818
2819 // if want to execute single index from multiple combined index its work left to
    right
2820 "indexName" : "dob.age_1_gender_1"
2821
2822 // it works fine, means it works with index scan
2823 > db.contacts.explain().find({'dob.age': 35})

```

```

2824 // it does not work properly means, it works with column scan
2825 > db.contacts.explain().find({'gender': 'male'})
2826
2827 // using indexes for sorting
2828 // this query also works like indexes
2829 // this also use index scan
2830 "indexName" : "dob.age_1_gender_1",
2831 > db.contacts.explain().find({'dob.age': 35}).sort({'gender': 1})
2832
2833 // mongo db reserves 32mb for fetched documents when using sort
2834
2835 // understanding the default index and find how many indexes into documents
2836 > db.contacts.getIndexes()
2837 [
2838   {
2839     "v" : 2,
2840     "key" : {
2841       "_id" : 1
2842     },
2843     "name" : "_id_",
2844     "ns" : "contactData.contacts"
2845   },
2846   {
2847     "v" : 2,
2848     "key" : {
2849       "dob.age" : 1,
2850       "gender" : 1
2851     },
2852     "name" : "dob.age_1_gender_1",
2853     "ns" : "contactData.contacts"
2854   }
2855 ]
2856 -----configuring Indexes-----
2857 // every indexes _id is a unique by default
2858 // can not add same value into same document
2859 > db.contacts.createIndex({email: 1}, {unique: true})
2860 {
2861   "ok" : 0,
2862   "errmsg" : "E11000 duplicate key error collection: contactData.contacts index:
email_1 dup key: { : \"abigail.clark@example.com\" }",
2863   "code" : 11000,
2864   "codeName" : "DuplicateKey"
2865 }
2866 > db.contacts.find({email: 'abigail.clark@example.com'}).count()
2867 // by checking can find unique value exist or not
2868
2869 // Understanding Partial Filters
2870 > db.contacts.dropIndex({'dob.age': 1, gender: 1})
2871
2872 // now create is an index on age, not on gender but on age but only for elements
where the underlying document is for a male.
2873 > db.contacts.createIndex({'dob.age': 1}, {partialFilterExpression:{gender: 'male'}})
2874 {
2875   "createdCollectionAutomatically" : false,
2876   "numIndexesBefore" : 1,
2877   "numIndexesAfter" : 2,
2878   "ok" : 1
2879 }
2880
2881 // can also use age query
2882 > db.contacts.createIndex({'dob.age': 1}, {partialFilterExpression:{'dob.age':
{$gt: 60}}})
2883 // this does not work. because as a partial index we have to also use gender
2884 > db.contacts.explain().find({'dob.age': {$gt: 60}})
2885 // this works with index scan
2886 > db.contacts.explain().find({'dob.age': {$gt: 60}, gender: 'male'})
2887
2888
2889 -----applying the Partial Index-----

```

```

2890 > db.users.insertMany([{name: 'Max', email: 'max@test.com'}, {name: 'Manu'}])
2891 // implementing unique key with email
2892 > db.users.createIndex({email: 1}, {unique: true})
2893
2894 // if now want to add new user without email, it says duplicate index error,
because no values store twice.
2895 > db.users.insertOne({name: 'Anna'})
2896
2897 // but a person could not have email
2898 > db.users.dropIndex({email: 1})
2899
2900 // now we create index a bit differently.
2901 > db.users.createIndex({email: 1}, {unique: true, partialFilterExpression: {email:
{$exists: true}}})
2902
2903 // now add user without email, it works
2904 > db.users.insertOne({name: 'Anna'})
2905
2906 // now we have three user one have email and others two without email
2907 > db.users.find().pretty()
2908
2909 // as partial index already created we can not add new user with same email
2910 > db.users.insertOne({name: 'Anna', email: 'max@test.com'})
2911
2912 // this section index options
2913 -----Understanding the Time-To-Live(TTL) index-----
2914
2915 // this works like session
2916 // clear data after some duration
2917 // self destroying data
2918 > db.sessions.insertOne({data: 'Sample data', createdAt: new Date()})
2919 > db.sessions.find().pretty()
2920
2921 // now add time to live index, can create with normal ascending text
2922 > db.sessions.createIndex({createdAt: 1})
2923 > db.sessions.dropIndex({createdAt: 1})
2924
2925 // add indexes with differently
2926 > db.sessions.createIndex({createdAt: 1}, {expireAfterSeconds: 10})
2927
2928 // after 10 seconds the document will be destroyed
2929 > db.sessions.find().pretty()
2930
2931 -----Query Diagnosis and Query Planing-----
2932
2933 // explain() it contains three types of parameter
2934 1. 'queryPlanner' --> Show Summary for Executed Query + Winning Plan
2935 2. 'executionsStats' --> Show Detailed Summary for Executed Query + Winning Plan +
Possibly Rejected Plans
2936 3. 'allPlanExecution' --> Show Detailed Summary for Executed Query + Winning Plan +
Winning Plan Decision Process
2937
2938 // Efficient Queries and Covered Queries
2939 // Milliseconds Process Time
2940 IXSCAN typically beats(1. of keys(in index) Examined 2. of Documents Examined 3. of
Documents Returns) COLLSCAN
2941
2942 // Understanding Covered Queries
2943 > db.customers.insertMany([{name: 'Max', age: 29, salary: 3000}, {name: 'Manu',
age: 30, salary: 4000}])
2944
2945 // creating index
2946 > db.customers.createIndex({name: 1})
2947 > db.customers.getIndexes()
2948
2949 // get info
2950 > db.customers.explain('executionStats').find({name: 'Max'})
2951
2952 // lets implement covered queries

```

```

2953 // if can optimize query, than have to reach that covered query state
2954 // useful when typically return the specific fields
2955 > db.customers.explain('executionStats').find({name: 'Max'}, {_id: 0, name: 1})
2956
2957 -----How mongoDB rejects a plan-----
2958
2959 // creating a compound index
2960 // order is important for compound index
2961 // name index here wouldn't make much sense
2962 // if age comes first, we can also filter just for age and take advantage of this
index.
2963 // if filtered for just name and didn't have that index, name could not be
supported by index.
2964
2965 // here we can use just age or combination of age and name.
2966 > db.customers.createIndex({age: 1, name: 1})
2967
2968 // let execute query, when execute query order does not matter in compound index
2969 > db.customers.explain().find({age: 30, name: 'Max'})
2970 > db.customers.explain().find({name: 'Max', age: 30})
2971
2972 // wining plan
2973 1. Approach 1
2974 2. Approach 2
2975 3. Approach 3 --> winning Plan --> Cached --> Cache --> but cache is not there
forever
2976
2977 // Clearing the Winning Plan from Cache
2978
2979 | 1. Write Threshold (currently 1,000)
2980 Stored Forever?--| 2. Index is Rebuilt
2981 | 3. Other Indexes are Added or Removed
2982 | 4. MongoDB Server is Restarted
2983
2984 > db.customers.insertOne({name:'Raju', age:22, salary: 1000})
2985
2986 // get details of all plan
2987 // here we get the all details of plan which be good and execution time
2988 > db.customers.explain('allPlansExecution').find({age: 30, name: 'Max'})
2989
2990 ----- Using Multi Key Indexes -----
2991
2992 // insert new data into new table
2993 > db.contactsinfo.insertOne({name:'Max', hobbies:['Cooking', 'Sports'], assress:
[{'street: 'Main Street'}, {'street: 'Second Street'}]})
2994
2995 > db.contactsinfo.findOne()
2996 // create an index
2997 > db.contactsinfo.createIndex({hobbies: 1})
2998 > db.contactsinfo.find({hobbies: 'Sports'}).pretty()
2999
3000 // execute explain
3001 // here multi key is true, it is created when documents into array
3002 > db.contactsinfo.explain('executionStats').find({hobbies: 'Sports'})
3003
3004 // lets create another index
3005 > db.contactsinfo.createIndex({addresses: 1})
3006
3007 // here index does not work,cause it does not work on nested documents query
3008 > db.contactsinfo.explain('executionStats').find({'addresses.street': 'Main Street'})
3009
3010 // it works when query like
3011 // Basically it works like normal
3012 > db.contactsinfo.explain('executionStats').find({addresses: {'street: 'Main
Street'}}})
3013
3014 // if the index is created like then it works, it also have a multi key index
3015 > db.contactsinfo.createIndex({'addresses.street': 1})
3016 // this is now index scan

```

```

3017 > db.contactsinfo.explain('executionStats').find({'addresses.street': 'Main Street'})
3018
3019 // Still multi key index is super helpful if have queries that regularly target
array values or even nested values or values in an embedded document in arrays.
3020 // There are a couple of restrictions or one important restriction to be precise
when using multi key indexes
3021
3022 // create a multi key compound index, it is also possible, when have one multi key
3023 > db.contactsinfo.createIndex({name: 1, hobbies: 1})
3024
3025 // but parallel arrays can not create multiple compound index
3026 > db.contactsinfo.createIndex({addresses: 1, hobbies: 1})
3027
3028 ----- Understanding 'text' indexes-----
3029
3030 // this is a special type of multi key index
3031 this product is a must-buy for all fans of modern fiction!
3032 // from the sentence the text index : product,must,buy,fans,modern,fiction
3033 // if an array of single words or array of keywords essentially to search text.
3034
3035 // create a new collections
3036 > 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
and it is pretty awesome'}])
3037
3038 // create an index
3039 // this is a single field index and can search with exact text
3040 > db.products.createIndex({description: 1})
3041
3042 // to create text index to split the sentence
3043 // so drop the previous index
3044 > db.products.dropIndex({description: 1})
3045 // create text index --> special kind of index
3046 // in text index remove all the stop words and store all the keyword into array
essentially
3047 > db.products.createIndex({description: 'text'})
3048
3049 // Now might be wondering why do not need to specify the field in which want to
search pretty expensive as can imagine.
3050 // if have a lot of long text that has to be split up,don't want to do this like 10
times per collection and therefore,only have one text index where this could look
into.
3051
3052 // can actually merge multiple fields into one text index.
3053 // everything is stored as lowercase.
3054 > db.products.find({'$text': {'$search': 'awesome'}}).pretty()
3055 {
3056   "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3057   "title" : "Red T-Shirt",
3058   "description" : "This T-Shirt is red and it is pretty awesome"
3059 }
3060 {
3061   "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3062   "title" : "A book",
3063   "description" : "This is an awesome book about a young artist!"
3064 }
3065
3066 > db.products.find({'$text': {'$search': 'book'}}).pretty()
3067 {
3068   "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3069   "title" : "A book",
3070   "description" : "This is an awesome book about a young artist!"
3071 }
3072
3073 // here red into second document and book into first document
3074 > db.products.find({'$text': {'$search': 'red book'}}).pretty()
3075 {
3076   "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3077   "title" : "A book",

```



```

3078         "description" : "This is an awesome book about a young artist!"
3079     }
3080     {
3081         "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3082         "title" : "Red T-Shirt",
3083         "description" : "This T-Shirt is red and it is pretty awesome"
3084     }
3085
3086 // can search with exactly phrase
3087 > db.products.find({$text: {$search: "\"red book\""}}).pretty()
3088 > db.products.find({$text: {$search: "\"awesome book\""}}).pretty()
3089 {
3090     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3091     "title" : "A book",
3092     "description" : "This is an awesome book about a young artist!"
3093 }
3094
3095 -----Text Indexes Sorting-----
3096
3097 // it works in new version automatically
3098 > db.products.find({$text: {$search: "awesome t-shirt"}}).pretty()
3099 {
3100     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3101     "title" : "Red T-Shirt",
3102     "description" : "This T-Shirt is red and it is pretty awesome"
3103 }
3104 {
3105     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3106     "title" : "A book",
3107     "description" : "This is an awesome book about a young artist!"
3108 }
3109
3110 // but in previous version
3111 > db.products.find({$text: {$search: "awesome t-shirt"}}).pretty()
3112 {
3113     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3114     "title" : "A book",
3115     "description" : "This is an awesome book about a young artist!"
3116 }
3117 {
3118     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3119     "title" : "Red T-Shirt",
3120     "description" : "This T-Shirt is red and it is pretty awesome"
3121 }
3122
3123 // lets add sorting query. in this query check how many words match with each
3124 // documents.
3125 // score increase with the number of matching words
3126 // which score is higher comes into first position
3127 > db.products.find({$text: {$search: "awesome t-shirt"}}, {score: {$meta:
3128 'textScore'}}).pretty()
3129 {
3130     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3131     "title" : "A book",
3132     "description" : "This is an awesome book about a young artist!",
3133     "score" : 0.6
3134 }
3135 {
3136     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3137     "title" : "Red T-Shirt",
3138     "description" : "This T-Shirt is red and it is pretty awesome",
3139     "score" : 1.7999999999999998
3140 }
3141
3142 // if sort does not work automatically add sort function and sort by score.
3143 > db.products.find({$text: {$search: "awesome t-shirt"}}, {score: {$meta:
3144 'textScore'}}).sort({score: {$meta: 'textScore'}}).pretty()

```

```

3144         "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3145         "title" : "Red T-Shirt",
3146         "description" : "This T-Shirt is red and it is pretty awesome",
3147         "score" : 1.7999999999999998
3148     }
3149 {
3150     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3151     "title" : "A book",
3152     "description" : "This is an awesome book about a young artist!",
3153     "score" : 0.6
3154 }
3155
3156 -----Combining Text Indexes-----
3157
3158 > db.products.getIndexes()
3159 > db.products.findOne()
3160 {
3161     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3162     "title" : "A book",
3163     "description" : "This is an awesome book about a young artist!"
3164 }
3165
3166 // if we now add text indexes with title like that it would be an error.
3167 // already text index is added with description in the document.
3168 // index option conflict
3169 // in every document we can must add only one text index
3170 > db.products.createIndex({title: 'text'})
3171
3172 // can merge the text of multiple fields together into one text index.
3173 // now drop the previous description text index, dropping text index is little bit
// different
3174 // have include the text index name
3175 > db.products.dropIndex('description_text')
3176 { "nIndexesWas" : 2, "ok" : 1 }
3177
3178 // now add two fields like title and description to create combined text index
3179 > db.products.createIndex({title:'text',description: 'text'})
3180 // insert a new element
3181 > db.products.insertOne({title: 'A Ship', description: 'Floats perfectly!'})
3182 // let execute query
3183 > db.products.find({$text: {$search: 'ship'}})
3184 > db.products.find({$text: {$search: 'awesome'}}).pretty()
3185
3186 // search with multiple text
3187 > db.products.find({$text: {$search: 'ship t-shirt'}}).pretty()
3188 > db.products.find({$text: {$search: 'awesome'}}).pretty()
3189 {
3190     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3191     "title" : "Red T-Shirt",
3192     "description" : "This T-Shirt is red and it is pretty awesome"
3193 }
3194 {
3195     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3196     "title" : "A book",
3197     "description" : "This is an awesome book about a young artist!"
3198 }
3199
3200 -----Using Text Index To Exclude Words-----
3201
3202 // to exclude words in search just add '-' before word
3203 // here want to search awesome but in the sentence if get awesome then exclude
t-shirt
3204 > db.products.find({$text: {$search: 'awesome -t-shirt'}}).pretty()
3205 {
3206     "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
3207     "title" : "A book",
3208     "description" : "This is an awesome book about a young artist!"
3209 }
3210

```

```

3211 -----Setting the Default Language Using Weights-----
3212
3213 // first dropping the previous text index
3214 > db.products.dropIndex('title_text_description_text')
3215 // pass some config
3216 > db.products.createIndex({title:'text',description: 'text'},{default_language:
3217 'german', weights: {title: 1, description: 10}})
3218 // can also work without weights, but without weights score value can be changed.
3219 > db.products.createIndex({title:'text',description: 'text'},{default_language:
3220 'english'})
3221
3222 > db.products.find({$text: {$search: '', $language: 'german'}}).pretty()
3223 // caseSensitive default is false
3224 > db.products.find({$text: {$search: '', $caseSensitive: true}}).pretty()
3225 > db.products.createIndex({title:'text',description: 'text'},{default_language:
3226 'english', weights: {title: 1, description: 10}})
3227
3228 // execute query
3229 > db.products.find({$text: {$search: 'red',}}).pretty()
3230 {
3231   "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3232   "title" : "Red T-Shirt",
3233   "description" : "This T-Shirt is red and it is pretty awesome"
3234 }
3235
3236 > db.products.find({$text: {$search: 'red'}},{score: {$meta: 'textScore'}}).pretty()
3237 {
3238   "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
3239   "title" : "Red T-Shirt",
3240   "description" : "This T-Shirt is red and it is pretty awesome",
3241   "score" : 6.666666666666667
3242 }
3243
3244 ----- Building Indexes-----
3245
3246 1. Foreground:
3247   a) Collection is locked during index creation.
3248   b) Faster
3249
3250 2. Background
3251   a) Collection is accessible during index creation.
3252   b) Slower
3253
3254 // In the previous, discussed about Foreground index(basically access from core db)
3255 // now create an index that basically a Background index
3256 // first discuss why Background index needs
3257
3258 > use credit
3259 switched to db credit
3260 > show collections
3261 ratings
3262 > db.ratings.find().count()
3263 1000000
3264
3265 > db.ratings.findOne()
3266
3267 // create an index with the age
3268 // here time is important cause documents size 100000
3269 > db.ratings.createIndex({age: 1})
3270
3271 // when creating an index into large scale documents or even a complex documents,
3272 db or documents is locked for a few seconds or couple of minutes
3273 // specially text indexes also need more time
3274 // so this is not an alternative production database
3275
3276 // after creating index if want to insert a new document into a large scale
3277 document then it also take a few lengthy time
3278 > db.ratings.insertOne({person_id: 'a39djd', score: 55.2211, age: 90})
3279 {
3280   "acknowledged" : true,

```

```

3275         "insertedId" : ObjectId("5f2ced58f48b8c5c77285c65")
3276     }
3277
3278     // let examine the query
3279     > db.ratings.explain('executionStats').find({age: {$gt: 80}})
3280     "executionTimeMillis" : 156
3281     > db.ratings.find({age: {$gt: 80}}).count()
3282     99792
3283
3284     // let drop the index
3285     > db.ratings.dropIndex({age: 1})
3286
3287     // let execute previous query
3288     > db.ratings.explain('executionStats').find({age: {$gt: 80}})
3289     "executionTimeMillis" : 367
3290
3291
3292     // let create a Background index
3293     // in Background index it takes a second argument
3294     // background default is false
3295     // so we have to set background to true
3296     // and it's created immediately
3297     > db.ratings.createIndex({age: 1}, {background: true})
3298
3299     // it happened in the background without locking the collection
3300
3301
3302
3303
3304
3305
3306
3307
3308
3309
3310
3311

```

---

## 11-working-with-geospatial-data

---

```

3315
3316     ----- Working with geo-spatial data-----
3317
3318     // Storing Gea-spatial data in Geo-JSON Format
3319     // Querying Gea-spatial Data
3320
3321     // GeoJson value is a embedded document, it contains two fields
3322     1. type --> specifies the GeoJson object type
3323     2. coordinates --> two values [longitude, latitude] in this format
3324
3325     // some operations ($near) require such an index.
3326     // but other operations like $geoWithin does not require index.
3327     // still can be used to speed up queries
3328
3329     GeoJson object Type: 1. Point 2. Line String 3. MultiLineString 4. Polygon 5.
    MultiPolygon 6. MultiPoint 7. GeometryCollection
3330
3331     // $GeoSpatial Queries : $near, $geoWithin, $geoIntersects
3332     // GeoSpatial queries work with GeoJson data
3333
3334     > use awesomeplaces
3335     switched to db awesomeplaces
3336     > db.places.insertOne({name: 'California Academy of
    Science', location: {type: 'Point', coordinates: [-122.4724356, 37.7672544]}})
3337
3338     // this is the GeoJson object
3339     > db.places.findOne()

```

```

3340 {
3341     "_id" : ObjectId("5f2f7ac7f82aee4b45288303"),
3342     "name" : "California Academy of Science",
3343     "location" : {
3344         "type" : "Point",
3345         "coordinates" : [
3346             -122.4724356,
3347             37.7672544
3348         ]
3349     }
3350 }
3351
3352 // let execute query into GeoJson data
3353 // to get our current location we have to use webApi or mobile api have to use any
other process so that user's can locate themselves
3354
3355 -----Finding nearest places from current location-----
3356
3357 // i want to find some places near my current location (sherpur home)
3358 // let's this is my location
3359 latitude : 25.0218715, longitude : 90.0106577
3360
3361 // Sherpur Government University College position --> latitude: 25.017493,longitude
: 90.011495
3362 > db.places.insertOne({name:'Sherpur Government University
College',location:{type:'Point',coordinates:[90.011495,25.017493]}})
3363
3364 // certain radius
3365 // first we have to create GeoSpatial index to track the distance
3366 // here the index name is '2dsphere'
3367 > db.places.createIndex({location: '2dsphere'})
3368
3369 // let find my nearest places
3370 > db.places.find({location: {$near: {$geometry: {type: 'Point', coordinates:
[90.0106577, 25.0218715]}}}}).pretty()
3371 {
3372     "_id" : ObjectId("5f2f81bef82aee4b45288304"),
3373     "name" : "Sherpur Government University College",
3374     "location" : {
3375         "type" : "Point",
3376         "coordinates" : [
3377             90.011495,
3378             25.017493
3379         ]
3380     }
3381 }
3382 {
3383     "_id" : ObjectId("5f2f7ac7f82aee4b45288303"),
3384     "name" : "California Academy of Science",
3385     "location" : {
3386         "type" : "Point",
3387         "coordinates" : [
3388             -122.4724356,
3389             37.7672544
3390         ]
3391     }
3392 }
3393
3394 // we can add also max and min distance into the query
3395 > db.places.find({location: {$near: {$geometry: {type: 'Point', coordinates:
[90.0106577, 25.0218715]}}}, $maxDistance: 30, $minDistance: 10}}).pretty()
3396 // we don't get any value, cause no places found with the distance that gives
3397
3398 // i calculate the distance between my current location and Sherpur Government
University College is 200.00m to 350m approximate
3399
3400 // so we have to add max distance a little bit large
3401
3402 > db.places.find({location: {$near: {$geometry: {type: 'Point', coordinates:

```

```

3403 [90.0106577, 25.0218715]], $maxDistance: 500, $minDistance: 20}}).pretty()
3404 {
3405   "_id" : ObjectId("5f2f81bef82aee4b45288304"),
3406   "name" : "Sherpur Government University College",
3407   "location" : {
3408     "type" : "Point",
3409     "coordinates" : [
3410       90.011495,
3411       25.017493
3412     ]
3413   }
3414 }
3415 -----Finding Points inside a covered area-----
3416
3417 // want to find all coordinates around the area
3418 // could be sphere, any polygon
3419 // which points are inside of the area.
3420 // here consider 4 points
3421 // previous California Academy of Science point also
3422
3423 // insert points
3424 > db.places.insertOne({name:'Conservatory of
Flowers',location:{type:'Point',coordinates:[-122.4615748, 37.7701756]}})
3425 > db.places.insertOne({name:'Golden Gate Park
Tennis',location:{type:'Point',coordinates:[-122.4593702, 37.7705046]}})
3426 >
db.places.insertOne({name:'Nopa',location:{type:'Point',coordinates:[-122.4389058,
37.7747415]}})
3427
3428 // show all documents
3429 > db.places.find().pretty()
3430
3431 // lets draw a polygon with the four points to check the area points exist in
database that covered by the area
3432 > const p1 = [-122.4547, 37.77473]
3433 > p1
3434 [ -122.4547, 37.77473 ]
3435 > const p2 = [-122.45303, 37.76641]
3436 > const p3 = [-122.51026, 37.76411]
3437 > const p4 = [-122.51088, 37.77131]
3438
3439 // here do not use $near instead use $geoWithin -->this can help to find all
elements within a certain shape, within a certain object, typically something like
polygon
3440 // have to also add p1(first corner) again, cause to complete the polygon and also
close the polygon
3441
3442 p1 ----- p2
3443 |           |
3444 |           |
3445 |           |
3446 p4 ----- p3
3447
3448 > db.places.find({location: {$geoWithin: {$geometry: {type: 'Polygon', coordinates:
[[p1, p2, p3, p4, p1]]}}}).pretty()
3449 {
3450   "_id" : ObjectId("5f2f9194f82aee4b45288305"),
3451   "name" : "Conservatory of Flowers",
3452   "location" : {
3453     "type" : "Point",
3454     "coordinates" : [
3455       -122.4615748,
3456       37.7701756
3457     ]
3458   }
3459 }
3460 {
3461   "_id" : ObjectId("5f2f931ff82aee4b45288306"),

```

```

    "name" : "Golden Gate Park Tennis",
    "location" : {
      "type" : "Point",
      "coordinates" : [
        -122.4593702,
        37.7705046
      ]
    }
  }
}
{
  "_id" : ObjectId("5f2f7ac7f82aee4b45288303"),
  "name" : "California Academy of Science",
  "location" : {
    "type" : "Point",
    "coordinates" : [
      -122.4724356,
      37.7672544
    ]
  }
}
}

-----Finding Out if a User is Inside a Specific Area-----

// Now another typical use case would be the opposite, that have an application
where want to find out whether the user is in a certain area.

// so don't want to find all places in an area but want to store a couple of
different areas potentially in the database

// let's say the neighborhoods of a city and then user sends some coordinates
because he located himself and want to find out in which neighborhood the user is.

// So essentially the same query as before, just the other way around.

// let insert data into new collections
> db.areas.insertOne({name: 'Golden Gate Park' ,area: {type: 'Polygon',
coordinates: [[p1, p2, p3, p4, p1]]}})
> db.areas.find().pretty()

// now create an index
> db.areas.createIndex({area: '2dsphere'})

// basically check $geoIntersects is true or false
// here .p --> means query point
p1 ----- p2
|           |
|           |
|       .p  |
|           |
p4 ----- p3

// Golden Gate Park is in -122.49089, 37.76992
// result --> here can see all points intersect
> db.areas.find({area: {$geoIntersects: {$geometry: {type: 'Point', coordinates:
[-122.49089, 37.76992]}}}}).pretty()

// check with the outside point
// result --> do not find any point
> db.areas.find({area: {$geoIntersects: {$geometry: {type: 'Point', coordinates:
[-122.48446, 37.77776]}}}}).pretty()

-----Finding Places Within a Certain Radius-----

// want to find all elements with unsorted order that are within certain radius
// want to find all places that are within a place or an area
// here have to use geoWithin not geoIntersects
// also can use $centerSphere instead of $geometry operator
// $centerSphere is a helpful operator that allows to quickly get a circle ar a point

```

```

3525 // so essentially it use a radius and a center and gives the whole circle around
3526 // $centerSphere first element--> the coordinates of the center of the circle want
3527 // $centerSphere second element --> a radius length with meter(m) , now interested
3528 // here use one kilometer
3529 // convert distance to radius, 6378.1 kilometer is a earth radius
3530 // here use places collection
3531 > db.places.find({location: {$geoWithin: {$centerSphere: [[-122.46203, 37.77286], 1
3532 / 6378.1]]}}).pretty()
3533 {
3534   "_id" : ObjectId("5f2f9194f82aee4b45288305"),
3535   "name" : "Conservatory of Flowers",
3536   "location" : {
3537     "type" : "Point",
3538     "coordinates" : [
3539       -122.4615748,
3540       37.7701756
3541     ]
3542   }
3543 }
3544 {
3545   "_id" : ObjectId("5f2f931ff82aee4b45288306"),
3546   "name" : "Golden Gate Park Tennis",
3547   "location" : {
3548     "type" : "Point",
3549     "coordinates" : [
3550       -122.4593702,
3551       37.7705046
3552     ]
3553   }
3554 }
3555 // California Academy of Science coordinates is falsely inserted, so update the query
3556 > db.places.updateOne({_id: ObjectId("5f2f7ac7f82aee4b45288303")}, {$set:
3557 {location: {type: 'Point', coordinates: [-122.46636, 37.77014]}}})
3558 > db.places.find({location: {$geoWithin: {$centerSphere: [[-122.46203, 37.77286], 1
3559 / 6378.1]]}}).pretty()
3560 {
3561   "_id" : ObjectId("5f2f7ac7f82aee4b45288303"),
3562   "name" : "California Academy of Science",
3563   "location" : {
3564     "type" : "Point",
3565     "coordinates" : [
3566       -122.46636,
3567       37.77014
3568     ]
3569   }
3570 }
3571 {
3572   "_id" : ObjectId("5f2f9194f82aee4b45288305"),
3573   "name" : "Conservatory of Flowers",
3574   "location" : {
3575     "type" : "Point",
3576     "coordinates" : [
3577       -122.4615748,
3578       37.7701756
3579     ]
3580   }
3581 }
3582 {
3583   "_id" : ObjectId("5f2f931ff82aee4b45288306"),
3584   "name" : "Golden Gate Park Tennis",
3585   "location" : {
3586     "type" : "Point",
3587     "coordinates" : [
3588       -122.4593702,
3589       37.7705046

```



```

3588         ]
3589     }
3590 }
3591
3592 // result--> here get the data with unsorted order. To sort the data apply manual
3593 approach
3594 // $near is the solution of sorted list
3595
3596 -----
3597 -----
3598 12-understanding-the-aggregation-framework
3599 -----
3600 -----
3601
3602 -----Aggregation Framework-----
3603
3604 // Retrieving Data Efficiently and In a Structured way
3605
3606 What is aggregation Framework
3607 // pipeline stages
3608 Steps for find (follow top to down)
3609         Collection
3610         |
3611         { $match }
3612         |
3613         { $sort }
3614         |
3615         { $group }
3616         |
3617         { $project }
3618         |
3619         Output (List of Documents)
3620
3621 -----Short Description -----
3622
3623 // Stages and Operations
3624 1. There are plenty of available stages and operations can choose from
3625 2. Stages define the different steps of and data is funneled through
3626 3. Each stage receives the output of the last stage as input
3627 4. Operations can be used inside of stages to transform, limit or re-calculated
3628    data.
3629
3630 // Important Stages
3631 1. The most important stages are $match, $group, $project, $sort and $unwind etc.
3632 2. Whilst there are some common behaviors between find() filters + projection and
3633    $match + $project, the aggregation stages are more flexible.
3634
3635 mongoimport persons.json -d analytics -c persons --jsonArray
3636 > use analytics
3637 > db.persons.findOne()
3638
3639 // The aggregate method takes an array and it takes an array cause have to define a
3640 series of steps inside array.
3641
3642 > db.persons.aggregate([
3643 ... { $match: {gender: 'female'} }
3644 ... ]).pretty()
3645
3646 -----Understanding the Group Stage-----
3647
3648 // group --> group stage allows to group data by a certain fields or by multiple
3649 fields
3650 // have to add $ sign before selected query document
3651 // here accumulate by 1 --> increasing value -1 -> decreasing value
3652 // totalPersons is the value that how many person are into same state

```

```

3650 // _id is unique value
3651 // can not use group into find() method
3652 // "$location.state" --> means iterating every element
3653
3654 db.persons.aggregate([
3655     { $match: { gender: 'female' } },
3656     { $group: { _id: { state: "$location.state" }, totalPersons: { $sum: 1 }}}
3657 ]).pretty()
3658
3659 // this is group stage in action
3660 // here we get the data with unsorted order
3661 // can also be sorted
3662
3663 { "_id" : { "state" : "berkshire" }, "totalPersons" : 1 }
3664 { "_id" : { "state" : "michigan" }, "totalPersons" : 1 }
3665 { "_id" : { "state" : "county down" }, "totalPersons" : 1 }
3666 { "_id" : { "state" : "loiret" }, "totalPersons" : 1 }
3667 { "_id" : { "state" : "cornwall" }, "totalPersons" : 2 }
3668 { "_id" : { "state" : "sivas" }, "totalPersons" : 1 }
3669 { "_id" : { "state" : "uşak" }, "totalPersons" : 1 }
3670 { "_id" : { "state" : "sinop" }, "totalPersons" : 3 }
3671 { "_id" : { "state" : "marne" }, "totalPersons" : 1 }
3672 { "_id" : { "state" : "northumberland" }, "totalPersons" : 1 }
3673 { "_id" : { "state" : "leicestershire" }, "totalPersons" : 1 }
3674 { "_id" : { "state" : "puy-de-dôme" }, "totalPersons" : 1 }
3675 { "_id" : { "state" : "maryland" }, "totalPersons" : 1 }
3676 { "_id" : { "state" : "ardèche" }, "totalPersons" : 1 }
3677 { "_id" : { "state" : "ankara" }, "totalPersons" : 3 }
3678 { "_id" : { "state" : "dordogne" }, "totalPersons" : 1 }
3679 { "_id" : { "state" : "antalya" }, "totalPersons" : 1 }
3680 { "_id" : { "state" : "corrèze" }, "totalPersons" : 1 }
3681 { "_id" : { "state" : "ardennes" }, "totalPersons" : 1 }
3682 { "_id" : { "state" : "bas-rhin" }, "totalPersons" : 2 }
3683 Type "it" for more
3684
3685
3686 // to check aggregation function work correctly
3687 > db.persons.find({'location.state': 'sinop', gender: 'female'}).count()
3688 3
3689
3690 // let also sort the group stage values according to totalPersons when execute query
3691 // sorting done from to previous stage
3692 > db.persons.aggregate([ { $match: { gender: 'female' } }, { $group: { _id:
{ state: "$location.state" }, totalPersons: { $sum: 1 } } }, { $sort: {
totalPersons: -1 } } ]).pretty()
3693 { "_id" : { "state" : "midtjylland" }, "totalPersons" : 33 }
3694 { "_id" : { "state" : "nordjylland" }, "totalPersons" : 27 }
3695 { "_id" : { "state" : "new south wales" }, "totalPersons" : 24 }
3696 {
3697     "_id" : {
3698         "state" : "australian capital territory"
3699     },
3700     "totalPersons" : 24
3701 }
3702 { "_id" : { "state" : "syddanmark" }, "totalPersons" : 24 }
3703 { "_id" : { "state" : "south australia" }, "totalPersons" : 22 }
3704 { "_id" : { "state" : "hovedstaden" }, "totalPersons" : 21 }
3705 { "_id" : { "state" : "danmark" }, "totalPersons" : 21 }
3706 { "_id" : { "state" : "queensland" }, "totalPersons" : 20 }
3707 { "_id" : { "state" : "overijssel" }, "totalPersons" : 20 }
3708 { "_id" : { "state" : "sjælland" }, "totalPersons" : 19 }
3709 { "_id" : { "state" : "nova scotia" }, "totalPersons" : 17 }
3710 { "_id" : { "state" : "canterbury" }, "totalPersons" : 16 }
3711 { "_id" : { "state" : "northwest territories" }, "totalPersons" : 16 }
3712 { "_id" : { "state" : " gelderland" }, "totalPersons" : 16 }
3713 { "_id" : { "state" : "yukon" }, "totalPersons" : 16 }
3714 { "_id" : { "state" : "bayern" }, "totalPersons" : 15 }
3715 { "_id" : { "state" : "northern territory" }, "totalPersons" : 15 }
3716 { "_id" : { "state" : "tasmania" }, "totalPersons" : 15 }

```

```

3717 { "_id" : { "state" : "noord-brabant" }, "totalPersons" : 14 }
3718 Type "it" for more
3719
3720 // check if answer is correctly
3721 > db.persons.find({'location.state': 'midtjylland', gender: 'female'}).count()
3722 33
3723
3724 ----- Working with Project Stage -----
3725
3726 // project works in the same way as the projection works in the find method
3727 "gender" : "male",
3728   "name" : {
3729     "title" : "mr",
3730     "first" : "harvey",
3731     "last" : "chambers"
3732   },
3733 // full list to all
3734 // want to convert name into one document
3735 // project does not group multiple documents together, its just transform every
    single document
3736
3737 > db.persons.aggregate([
3738   { $project: { _id: 0, gender: 1, fullName: { $concat: ['$name.first', '
    ', '$name.last'] } } }
3739 ]).pretty()
3740
3741 // now want to first and last name start with Uppercase letter
3742 > db.persons.aggregate([
3743   {
3744     $project: {
3745       _id: 0,
3746       gender: 1,
3747       fullName: {
3748         $concat: [{ $toUpper: '$name.first'}, ' ', { $toUpper: '$name.last'}]
3749       }
3750     }
3751   }
3752 ]).pretty()
3753
3754 // $substrCP --> substring part
3755 // 0 -> means starting index
3756 // 1 -> means how much character (length)
3757 > db.persons.aggregate([
3758   {
3759     $project: {
3760       _id: 0,
3761       gender: 1,
3762       fullName: {
3763         $concat: [
3764           { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
3765           ' ',
3766           { $toUpper: { $substrCP: ['$name.last', 0, 1] } }
3767         ]
3768       }
3769     }
3770   }
3771 ]).pretty()
3772
3773 // the final output
3774 > db.persons.aggregate([
3775   {
3776     $project: {
3777       _id: 0,
3778       gender: 1,
3779       fullName: {
3780         $concat: [
3781           { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
3782           { $substrCP: ['$name.first', 1, { $subtract: [{ $strLenCP:
    '$name.first' }, 1] } ] } ],

```

```

3783         ' ',
3784         { $toUpper: { $substrCP: ['$name.last', 0, 1] } },
3785         { $substrCP: ['$name.last', 1, { $subtract: [{ $strLenCP:
           '$name.last' }, 1] }] },
3786     ]
3787 }
3788 }
3789 }
3790 ]).pretty()
3791
3792
3793 -----
3794
3794 12-understanding-the-aggregation-framework
3795 using-the-aggregation-framework (part-2)
3796 -----
3797
3798
3799 ----- Turning the Location Into a geoJSON Object-----
3800
3801 // using multiple aggregate function to get the next value from previous
3802
3803 > db.persons.aggregate([
3804     {
3805         $project: {
3806             _id: 0,
3807             name: 1,
3808             email: 1,
3809             location: {
3810                 type: 'Point',
3811                 coordinates: [
3812                     '$location.coordinates.longitude',
3813                     '$location.coordinates.latitude',
3814                 ]
3815             }
3816         },
3817     },
3818     {
3819         $project: {
3820             email: 1,
3821             location: 1,
3822             gender: 1,
3823             fullName: {
3824                 $concat: [
3825                     { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
3826                     { $substrCP: ['$name.first', 1, { $subtract: [{ $strLenCP:
                       '$name.first' }, 1] }] },
3827                     ' ',
3828                     { $toUpper: { $substrCP: ['$name.last', 0, 1] } },
3829                     { $substrCP: ['$name.last', 1, { $subtract: [{ $strLenCP:
                       '$name.last' }, 1] }] },
3830                 ]
3831             }
3832         }
3833     }
3834 ]).pretty()
3835
3836 {
3837     "location" : {
3838         "type" : "Point",
3839         "coordinates" : [
3840             "168.9462",
3841             "-22.5329"
3842         ]
3843     },
3844     "email" : "harvey.chambers@example.com",
3845     "fullName" : "Harvey Chambers"
3846 }

```

```

3847
3848
3849 // here getting coordinates as a string, so have to convert into number
3850 > db.persons.aggregate([
3851   {
3852     $project: {
3853       _id: 0,
3854       name: 1,
3855       email: 1,
3856       location: {
3857         type: 'Point',
3858         coordinates: [
3859           {
3860             $convert: {
3861               input: '$location.coordinates.longitude',
3862               to: 'double',
3863               onError: 0.0,
3864               onNull: 0.0
3865             }
3866           },
3867           {
3868             $convert: {
3869               input: '$location.coordinates.latitude',
3870               to: 'double',
3871               onError: 0.0,
3872               onNull: 0.0
3873             }
3874           }
3875         ]
3876       }
3877     },
3878   },
3879   {
3880     $project: {
3881       email: 1,
3882       location: 1,
3883       gender: 1,
3884       fullName: {
3885         $concat: [
3886           {
3887             $toUpper: {
3888               $substrCP: ['$name.first', 0, 1]
3889             }
3890           }, {
3891             $substrCP: [
3892               '$name.first', 1, {
3893                 $subtract: [
3894                   { $strLenCP: '$name.first' }, 1
3895                 ]
3896             }
3897           ],
3898           ' ',
3899           {
3900             $toUpper: {
3901               $substrCP: ['$name.last', 0, 1]
3902             }
3903           },
3904           {
3905             $substrCP: [
3906               '$name.last', 1, {
3907                 $subtract: [
3908                   { $strLenCP: '$name.last' }, 1
3909                 ]
3910             }
3911           }
3912         ]
3913       }
3914     }
3915   ]

```

```

3916   }).pretty()
3917
3918   // transforming the BirthDate into data format
3919   db.persons.aggregate([
3920     {
3921       $project: {
3922         _id: 0,
3923         name: 1,
3924         email: 1,
3925         birthdate: {
3926           $convert: {
3927             input: '$dob.date',
3928             to: 'date',
3929             onError: 0.0,
3930             onNull: 0.0
3931           }
3932         },
3933         age: '$dob.age',
3934         location: {
3935           type: 'Point',
3936           coordinates: [
3937             {
3938               $convert: {
3939                 input: '$location.coordinates.longitude',
3940                 to: 'double',
3941                 onError: 0.0,
3942                 onNull: 0.0
3943               }
3944             },
3945             {
3946               $convert: {
3947                 input: '$location.coordinates.latitude',
3948                 to: 'double',
3949                 onError: 0.0,
3950                 onNull: 0.0
3951               }
3952             }
3953           ]
3954         }
3955       }
3956     },
3957     {
3958       $project: {
3959         email: 1,
3960         location: 1,
3961         gender: 1,
3962         birthdate: 1,
3963         age: 1,
3964         fullName: {
3965           $concat: [
3966             {
3967               $toUpper: {
3968                 $substrCP: ['$name.first', 0, 1]
3969               }
3970             }, {
3971               $substrCP: [
3972                 '$name.first', 1, {
3973                   $subtract: [
3974                     { $strlenCP: '$name.first' }, 1
3975                   ]
3976                 }
3977             ]
3978           },
3979           {
3980             $toUpper: {
3981               $substrCP: ['$name.last', 0, 1]
3982             }
3983           }
3984         ]

```



```

4053         ' ',
4054     {
4055         $toUpper: {
4056             $substrCP: ['$name.last', 0, 1]
4057         }
4058     },
4059     {
4060         $substrCP: [
4061             '$name.last', 1, {
4062                 $subtract: [
4063                     { $strLenCP: '$name.last' }, 1
4064                 ]
4065             }]
4066     }
4067 ]
4068 }
4069 }
4070 }
4071 ]).pretty()
4072
4073
4074

```

```

4075 -----
4076 12-understanding-the-aggregation-framework
4077 using-the-aggregation-framework(part-3)
4078 -----

```

```

4078 -----Understanding the ISO Week Year Operator-----
4079
4080
4081 // $isoWeekYear retrieves the year out of date
4082 db.persons.aggregate([
4083     {
4084         $project: {
4085             _id: 0,
4086             name: 1,
4087             email: 1,
4088             birthdate: { $toDate: '$dob.date' },
4089             age: '$dob.age',
4090             location: {
4091                 type: 'Point',
4092                 coordinates: [
4093                     {
4094                         $convert: {
4095                             input: '$location.coordinates.longitude',
4096                             to: 'double',
4097                             onError: 0.0,
4098                             onNull: 0.0
4099                         }
4100                     },
4101                 ],
4102             },
4103             $convert: {
4104                 input: '$location.coordinates.latitude',
4105                 to: 'double',
4106                 onError: 0.0,
4107                 onNull: 0.0
4108             }
4109         }
4110     ]
4111 }
4112 },
4113 {
4114     $project: {
4115         email: 1,
4116         location: 1,
4117         gender: 1,
4118         birthdate: 1,

```



```

4120         age: 1,
4121         fullName: {
4122             $concat: [
4123                 {
4124                     $toUpper: {
4125                         $substrCP: ['$name.first', 0, 1]
4126                     }
4127                 }, {
4128                     $substrCP: [
4129                         '$name.first', 1, {
4130                             $subtract: [
4131                                 { $strlenCP: '$name.first' }, 1
4132                             ]
4133                         }
4134                     ],
4135                     ' ',
4136                     {
4137                         $toUpper: {
4138                             $substrCP: ['$name.last', 0, 1]
4139                         }
4140                     },
4141                     {
4142                         $substrCP: [
4143                             '$name.last', 1, {
4144                                 $subtract: [
4145                                     { $strlenCP: '$name.last' }, 1
4146                                 ]
4147                             }
4148                         ]
4149                     }
4150                 ]
4151             }
4152         },
4153         { $group: { _id: { birthYear: { $isoWeekYear: '$birthdate' } }, numPersons: {
4154             $sum: 1 } } }
4155     ]).pretty()
4156
4157 // adding sort
4158 db.persons.aggregate([
4159     {
4160         $project: {
4161             _id: 0,
4162             name: 1,
4163             email: 1,
4164             birthdate: { $toDate: '$dob.date' },
4165             age: '$dob.age',
4166             location: {
4167                 type: 'Point',
4168                 coordinates: [
4169                     {
4170                         $convert: {
4171                             input: '$location.coordinates.longitude',
4172                             to: 'double',
4173                             onError: 0.0,
4174                             onNull: 0.0
4175                         }
4176                     },
4177                     {
4178                         $convert: {
4179                             input: '$location.coordinates.latitude',
4180                             to: 'double',
4181                             onError: 0.0,
4182                             onNull: 0.0
4183                         }
4184                     }
4185                 ]
4186             }
4187         }
4188     }
4189 ]

```

```

4188     },
4189     {
4190         $project: {
4191             email: 1,
4192             location: 1,
4193             gender: 1,
4194             birthdate: 1,
4195             age: 1,
4196             fullName: {
4197                 $concat: [
4198                     {
4199                         $toUpper: {
4200                             $substrCP: ['$name.first', 0, 1]
4201                         }
4202                     }, {
4203                         $substrCP: [
4204                             '$name.first', 1, {
4205                                 $subtract: [
4206                                     { $strLenCP: '$name.first' }, 1
4207                                 ]
4208                             }]
4209                     },
4210                     ' ',
4211                     {
4212                         $toUpper: {
4213                             $substrCP: ['$name.last', 0, 1]
4214                         }
4215                     },
4216                     {
4217                         $substrCP: [
4218                             '$name.last', 1, {
4219                                 $subtract: [
4220                                     { $strLenCP: '$name.last' }, 1
4221                                 ]
4222                             }]
4223                     }
4224                 ]
4225             }
4226         },
4227         { $group: { _id: { birthYear: { $isoWeekYear: '$birthdate' } }, numPersons: {
4228             $sum: 1 } } },
4229         { $sort: { numPersons: -1}}
4230     ]).pretty()
4231
4232     -----$group vs $project-----
4233
4234     $group :
4235     1. grouping multiple documents into one document.
4236     2. n:1
4237     3. have multiple documents and return one grouped by one or more categories.
4238     4. do things like summing, counting, averaging, build array and so on
4239
4240     $project:
4241     1. get one document and then will return one document, that one document we'll
4242        just have changed.
4243     2. 1:1
4244     3. transform a single document, add new fields and so on.
4245     4. have a one to one relation, include/exclude fields.
4246
4247     -----Pushing Elements Into Newly Created Arrays-----
4248
4249     // push Operator allows to push a new element into the all hobbies array for every
4250     incoming document.
4251
4252     > db.friends.aggregate([
4253     ...     { $group: { _id: { age: '$age' }, allHobbies: {$push: '$hobbies'}}}
4254     ... ]).pretty()
4255     {

```

```

4254     "_id" : {
4255         "age" : 29
4256     },
4257     "allHobbies" : [
4258         [
4259             "Sports",
4260             "Cooking"
4261         ],
4262         [
4263             "Cooking",
4264             "Skiing"
4265         ]
4266     ]
4267 }
4268 {
4269     "_id" : {
4270         "age" : 30
4271     },
4272     "allHobbies" : [
4273         [
4274             "Eating",
4275             "Data Analytics"
4276         ]
4277     ]
4278 }
4279
4280 -----Understanding the unwind Stage-----
4281
4282 // do not want to insert into nested array
4283 // The unwind stage is always a great stage when have an array of which want to
pull out the elements.
4284
4285 > db.friends.aggregate([
4286     { $unwind: '$hobbies' }
4287 ]).pretty()
4288 // result -> every array element has one document
4289
4290 // now adding group to every document according to age
4291 > db.friends.aggregate([
4292     ... { $unwind: '$hobbies' },
4293     ... { $group: { _id: { age: '$age' }, allHobbies: { $push: '$hobbies' } } }
4294     ... ]).pretty()
4295 {
4296     "_id" : {
4297         "age" : 29
4298     },
4299     "allHobbies" : [
4300         "Sports",
4301         "Cooking",
4302         "Cooking",
4303         "Skiing"
4304     ]
4305 }
4306 {
4307     "_id" : {
4308         "age" : 30
4309     },
4310     "allHobbies" : [
4311         "Eating",
4312         "Data Analytics"
4313     ]
4314 }
4315
4316
4317
4318
4319 -----
-----

```



```

4389         examScore: {
4390             $slice: ['$examScores', 2]
4391         }
4392     }
4393 }
4394 ]).pretty()
4395
4396 // -1 means last
4397
4398 > db.friends.aggregate([
4399     {
4400         $project: {
4401             _id: 0,
4402             examScore: {
4403                 $slice: ['$examScores', -1]
4404             }
4405         }
4406     }
4407 ]).pretty()
4408 { "examScore" : [ { "difficulty" : 3, "score" : 88.5 } ] }
4409 { "examScore" : [ { "difficulty" : 5, "score" : 53.1 } ] }
4410 { "examScore" : [ { "difficulty" : 6, "score" : 61.5 } ] }
4411
4412 // last two scores
4413 > db.friends.aggregate([
4414     {
4415         $project: {
4416             _id: 0,
4417             examScore: {
4418                 $slice: ['$examScores', -2]
4419             }
4420         }
4421     }
4422 ]).pretty()
4423
4424 // start at position two and give one element
4425 > db.friends.aggregate([
4426     {
4427         $project: {
4428             _id: 0,
4429             examScore: {
4430                 $slice: ['$examScores', 2, 1]
4431             }
4432         }
4433     }
4434 ]).pretty()
4435
4436 -----Getting the length of and array-----
4437
4438 // $size calculate the length of an array
4439
4440 > db.friends.aggregate([
4441     {
4442         $project: {
4443             _id: 0,
4444             numScores: {
4445                 $size: '$examScores'
4446             }
4447         }
4448     }
4449 ]).pretty()
4450
4451
4452
4453 -----Using the Filter Operator-----
4454
4455 // $filter Operator allows to filter out certain elements an array and only return
the data according to condition
4456 // filter score so the greater than 60

```

```

4457 // here sc is a temporary variable for using condition
4458 // sc is a temporary variable of examScores but filter function executes over and
over again all fields
4459 // so have to use two dollar sign
4460 // $cond --> condition
4461
4462 db.friends.aggregate([
4463   {
4464     $project: {
4465       _id: 0,
4466       scores: {
4467         $filter: {
4468           input: '$examScores',
4469           as: 'sc',
4470           cond: {
4471             $gt: ['$sc.score', 60]
4472           }
4473         }
4474       }
4475     }
4476   }
4477 ]).pretty()
4478 {
4479   "scores" : [
4480     {
4481       "difficulty" : 6,
4482       "score" : 62.1
4483     },
4484     {
4485       "difficulty" : 3,
4486       "score" : 88.5
4487     }
4488   ]
4489 }
4490 { "scores" : [ { "difficulty" : 2, "score" : 74.3 } ] }
4491 {
4492   "scores" : [
4493     {
4494       "difficulty" : 3,
4495       "score" : 75.1
4496     },
4497     {
4498       "difficulty" : 6,
4499       "score" : 61.5
4500     }
4501   ]
4502 }
4503
4504 -----Applying Multiple Operations to our Array-----
4505
4506 wanted to transform our friend objects such that only output the highest exam score
4507
4508 > db.friends.aggregate([
4509   { $unwind: '$examScores' },
4510   { $sort: { 'examScores.score': -1 } }
4511 ]).pretty()
4512
4513 // can do same thing by projection
4514
4515 > db.friends.aggregate([
4516   { $unwind: '$examScores' },
4517   { $project: { _id: 1, name: 1, age: 1, score: '$examScores.score' } },
4518   { $sort: { score: -1 } },
4519   { $group: { _id: '$_id', maxScore: { $max: '$score' } } }
4520 ]).pretty()
4521
4522 // group by id but can also add anything
4523 // if can group by name, it is bad choice. cause name can be duplicate
4524 > db.friends.aggregate([

```

```

4525     { $unwind: '$examScores' },
4526     { $project: { _id: 1, name: 1, age: 1, score: '$examScores.score' } },
4527     { $group: { _id: '$_id', maxScore: { $max: '$score' } } }
4528   }).pretty()
4529
4530   { "_id" : ObjectId("5f318bb939e723820551436e"), "maxScore" : 74.3 }
4531   { "_id" : ObjectId("5f318bb939e723820551436f"), "maxScore" : 75.1 }
4532   { "_id" : ObjectId("5f318bb939e723820551436d"), "maxScore" : 88.5 }
4533
4534   // show the name and sort with descending order
4535   // use the first value encounter
4536
4537   // $first --> means want to get the name value
4538   > db.friends.aggregate([
4539     { $unwind: '$examScores' },
4540     { $project: { _id: 1, name: 1, age: 1, score: '$examScores.score' } },
4541     { $group: { _id: '$_id', name: { $first: '$name' }, maxScore: { $max: '$score' } } },
4542     { $sort: { maxScore: -1 } }
4543   ]).pretty()
4544
4545   {
4546     "_id" : ObjectId("5f318bb939e723820551436d"),
4547     "name" : "Max",
4548     "maxScore" : 88.5
4549   }
4550   {
4551     "_id" : ObjectId("5f318bb939e723820551436f"),
4552     "name" : "Maria",
4553     "maxScore" : 75.1
4554   }
4555   {
4556     "_id" : ObjectId("5f318bb939e723820551436e"),
4557     "name" : "Manu",
4558     "maxScore" : 74.3
4559   }
4560   }
4561
4562   > db.friends.aggregate([
4563     { $unwind: '$examScores' },
4564     { $project: { _id: 1, name: 1, age: 1, score: '$examScores.score' } },
4565     { $group: { _id: '$_id', name: { $first: '$name' }, age: { $first: '$age' },
4566       maxScore: { $max: '$score' } } },
4567     { $sort: { maxScore: -1 } }
4568   ]).pretty()
4569
4570
4571
4572

```

---

## 12-understanding-the-aggregation-framework

### using-the-aggregation-framework (part-5)

---

```

4576
4577
4578   -----Understanding bucket-----
4579
4580   // let's prepare a bucket stage
4581   // using bucket can create a different categories and filter
4582
4583   // boundaries means range/levels like 0-18, 18-30,30-50, 50-80, 80-120
4584   // in every range first value execute not last value, 18-30 --> means with 18 but
4585   // not exist 30
4586   > db.persons.aggregate([
4587     {
4588       $bucket: {
4589         groupBy: '$dob.age',

```

```

4589         boundaries: [0, 18, 30, 50, 80, 120],
4590         output: {
4591             numPersons: { $sum: 1 },
4592             average: { $avg: '$dob.age' },
4593         }
4594     }
4595 }
4596 ]).pretty()
4597
4598 // here we get the three bucket
4599 { "_id" : 18, "numPersons" : 868, "average" : 25.101382488479263 }
4600 { "_id" : 30, "numPersons" : 1828, "average" : 39.4917943107221 }
4601 { "_id" : 50, "numPersons" : 2304, "average" : 61.46440972222222 }
4602
4603 > db.persons.find({'dob.age': {$gt: 17, $lt: 30}}).count()
4604 868
4605
4606 > db.persons.find({'dob.age': {$gt: 49, $lt: 80}}).count()
4607 2304
4608
4609 > db.persons.find({'dob.age': {$gt: 29, $lt: 50}}).count()
4610 1828
4611
4612
4613 // checking the validation
4614 // no data
4615 > db.persons.find({'dob.age': {$lt: 18}})
4616 > db.persons.find({'dob.age': {$gt: 80}})
4617 > db.persons.find({'dob.age': 80})
4618
4619 // adding more levels
4620
4621 > db.persons.aggregate([
4622     {
4623         $bucket: {
4624             groupBy: '$dob.age',
4625             boundaries: [18, 30, 40, 50, 60, 120],
4626             output: {
4627                 numPersons: { $sum: 1 },
4628                 average: { $avg: '$dob.age' },
4629             }
4630         }
4631     }
4632 ]).pretty()
4633
4634 // can also create a auto bucket by defining how many buckets want
4635 // almost have equal distributions
4636
4637 > db.persons.aggregate([
4638     {
4639         $bucketAuto: {
4640             groupBy: '$dob.age',
4641             buckets: 5,
4642             output: {
4643                 numPersons: { $sum: 1 },
4644                 average: { $avg: '$dob.age' },
4645             }
4646         }
4647     }
4648 ]).pretty()
4649
4650 {
4651     "_id" : {
4652         "min" : 21,
4653         "max" : 32
4654     },
4655     "numPersons" : 1042,
4656     "average" : 25.99616122840691
4657 }

```



```

4658 {
4659     "_id" : {
4660         "min" : 32,
4661         "max" : 43
4662     },
4663     "numPersons" : 1010,
4664     "average" : 36.97722772277228
4665 }
4666 {
4667     "_id" : {
4668         "min" : 43,
4669         "max" : 54
4670     },
4671     "numPersons" : 1033,
4672     "average" : 47.98838334946757
4673 }
4674 {
4675     "_id" : {
4676         "min" : 54,
4677         "max" : 65
4678     },
4679     "numPersons" : 1064,
4680     "average" : 58.99342105263158
4681 }
4682 {
4683     "_id" : {
4684         "min" : 65,
4685         "max" : 74
4686     },
4687     "numPersons" : 851,
4688     "average" : 69.11515863689776
4689 }
4690
4691 -----Diving into Additional Stages-----
4692
4693 // want to find the 10 users, the 10 persons with the oldest birth date, so the
4694 // lowest birth date
4695 > db.persons.aggregate([
4696     {
4697         $project: {
4698             _id: 0,
4699             name: 1,
4700             birthDate: {
4701                 $toDate: '$dob.date'
4702             }
4703         }
4704     }
4705 ]).pretty()
4706
4707 > db.persons.aggregate([
4708     { $project: { _id: 0, name: 1, birthDate: { $toDate: '$dob.date' } } },
4709     { $sort: { birthDate: 1 } },
4710     { $limit: 10 }
4711 ]).pretty()
4712
4713 // adding some extra
4714 > db.persons.aggregate([
4715     { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
4716         birthDate: { $toDate: '$dob.date' } } },
4717     { $sort: { birthDate: 1 } },
4718     { $limit: 10 }
4719 ]).pretty()
4720
4721 // skip first 10
4722 > db.persons.aggregate([
4723     { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
4724         birthDate: { $toDate: '$dob.date' } } },
4725     { $sort: { birthDate: 1 } },

```

```

4724     { $skip: 10 },
4725     { $limit: 10 }
4726   }).pretty()
4727
4728   // but after $skip into $limit it does not work
4729   > db.persons.aggregate([
4730     { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
4731       birthDate: { $toDate: '$dob.date' } } },
4732     { $sort: { birthDate: 1 } },
4733     { $limit: 10 },
4734     { $skip: 10 }
4735   ]).pretty()
4736
4737   // if add sort into last can see the different result
4738   > db.persons.aggregate([
4739     { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
4740       birthDate: { $toDate: '$dob.date' } } },
4741     { $limit: 10 },
4742     { $skip: 10 },
4743     { $sort: { birthDate: 1 } },
4744   ]).pretty()
4745
4746   // same also for $match
4747   > db.persons.aggregate([
4748     { $match: { gender: 'male' } },
4749     { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
4750       birthDate: { $toDate: '$dob.date' } } },
4751     { $skip: 10 },
4752     { $limit: 10 },
4753     { $sort: { birthDate: 1 } }
4754   ]).pretty()
4755
4756   // if $match add after the project without projection, we do not get any result
4757   > db.persons.aggregate([
4758     { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
4759       birthDate: { $toDate: '$dob.date' } } },
4760     { $sort: { birthDate: 1 } },
4761     { $match: { gender: 'male' } },
4762     { $skip: 10 },
4763     { $limit: 10 },
4764   ]).pretty()
4765
4766   // if gender add into projection phase then will get results
4767   > db.persons.aggregate([
4768     { $project: { _id: 0, gender: 1, name: { $concat: ['$name.first', ' ', '$name.last'] },
4769       birthDate: { $toDate: '$dob.date' } } },
4770     { $sort: { birthDate: 1 } },
4771     { $match: { gender: 'male' } },
4772     { $skip: 10 },
4773     { $limit: 10 },
4774   ]).pretty()
4775
4776   // but best is, use $match before $project
4777   > db.persons.aggregate([
4778     { $match: { gender: 'male' } },
4779     { $project: { _id: 0, gender: 1, name: { $concat: ['$name.first', ' ', '$name.last'] },
4780       birthDate: { $toDate: '$dob.date' } } },
4781     { $sort: { birthDate: 1 } },
4782     { $skip: 10 },
4783     { $limit: 10 },
4784   ]).pretty()
4785
4786   -----Writing Pipeline Results Into a New Collection-----
4787
4788   // by getting the output we can store into the another Collection
4789   // can do work with the out stage
4790   db.persons.aggregate([
4791     {
4792       $project: {

```

```

4787     _id: 0,
4788     name: 1,
4789     email: 1,
4790     birthdate: { $toDate: '$dob.date' },
4791     age: "$dob.age",
4792     location: {
4793         type: 'Point',
4794         coordinates: [
4795             {
4796                 $convert: {
4797                     input: '$location.coordinates.longitude',
4798                     to: 'double',
4799                     onError: 0.0,
4800                     onNull: 0.0
4801                 }
4802             },
4803             {
4804                 $convert: {
4805                     input: '$location.coordinates.latitude',
4806                     to: 'double',
4807                     onError: 0.0,
4808                     onNull: 0.0
4809                 }
4810             }
4811         ]
4812     }
4813 },
4814 {
4815     $project: {
4816         gender: 1,
4817         email: 1,
4818         location: 1,
4819         birthdate: 1,
4820         age: 1,
4821         fullName: {
4822             $concat: [
4823                 { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
4824                 {
4825                     $substrCP: [
4826                         '$name.first',
4827                         1,
4828                         { $subtract: [{ $strLenCP: '$name.first' }, 1] }
4829                     ]
4830                 },
4831                 ' ',
4832                 { $toUpper: { $substrCP: ['$name.last', 0, 1] } },
4833                 {
4834                     $substrCP: [
4835                         '$name.last',
4836                         1,
4837                         { $subtract: [{ $strLenCP: '$name.last' }, 1] }
4838                     ]
4839                 }
4840             ]
4841         }
4842     }
4843 },
4844 { $out: "transformedPersons" }
4845 ]).pretty();
4846
4847 -----Working with the geoNear Stage-----
4848
4849 // first create an index into the transformedPersons Collection
4850
4851 > db.transformedPersons.createIndex({location: '2dsphere'})
4852 // create geo location aggregation pipeline stages
4853
4854 // have to specify and that is the distance field, because geoNear will actually

```

also give us back the distance that is calculated between our point and the document

// geoNear, it has to be the first element in the pipeline because it needs to use that index and the first pipeline element is the only element with direct access to the collection, other pipeline stages just get the output of the previous pipeline stage, this is the only element with direct access to the collection.

// also can add query

```
> db.transformedPersons.aggregate([
  {
    $geoNear: {
      near: {
        type: 'Point',
        coordinates: [-18.4, -42.8]
      },
      maxDistance: 1000000,
      $limit: 10,
      query: { age: { $gt: 30 } },
      distanceField: 'distance'
    }
  }
]).pretty()
```

// can also add multiple pipeline stages

```
db.transformedPersons.aggregate([
  {
    $geoNear: {
      near: {
        type: 'Point',
        coordinates: [-18.4, -42.8]
      },
      maxDistance: 1000000,
      $limit: 10,
      query: { age: { $gt: 30 } },
      distanceField: 'distance'
    }
  },
  { $project: { _id: 1, email: 0, birthdate: 0 } },
  { $sort: { distanceField: 1 } }
]).pretty()
```

---

### 13-working-with-numeric-data

---

#### -----Working With Numeric Data-----

// numeric data is most important in scientific calculation  
// Number more complex than any other  
// 3 types of number in mongoDB. (Integers, Longs, Doubles)

Integers(int32) -->

1. Only full Numbers(+/- 2<sup>32</sup>).
2. Use for 'normal' Integers

Longs(int64) -->

1. Only full Numbers(+/- 2<sup>64</sup>).
2. Use for large Integers

Doubles(64bit) -->

1. Numbers with Decimal Places(Decimal values are approximated).
2. Use for floats where high precision is not required

High Precision Doubles(128bit) -->

1. Numbers with Decimal Places(Decimal values are stored with high precision(34

```

4918 decimal digits)).
4919 2. Use for floats where high precision is required
4920
4921 // in mongoDB driver is a javaScript based driver.
4922 // all numeric values stored as a double
4923
4924 > use numeric
4925 switched to db numeric
4926 > db.persons.insertOne({name: 'Max', age: 29})
4927 > db.persons().find()
4928
4929 -----Working with Int32-----
4930
4931 // here can see the size
4932 > db.persons.stats()
4933 "size" : 49,
4934 "count" : 1,
4935 > db.persons.deleteMany({})
4936
4937 // here can see that size is decrease
4938 > db.persons.insertOne({ age: 29})
4939 > db.persons.stats()
4940 "size" : 35,
4941 "count" : 1,
4942 > db.persons.deleteMany({})
4943
4944 // here also can see that size now also more decrease
4945 > db.persons.insertOne({ age: NumberInt(29)})
4946 > db.persons.insertOne({ age: NumberInt("29")})
4947 "size" : 31,
4948 "count" : 1,
4949
4950 -----Working with Int64-----
4951
4952 // here can the output is a wrong value
4953 > db.companies.insertOne({valuation: NumberInt('50000000000')})
4954 > db.companies.findOne()
4955 { "_id" : ObjectId("5f3e81e7d0209e4d3a0ec072"), "valuation" : -1539607552
4956
4957 // if decrease one value then it works
4958 > db.companies.insertOne({valuation: NumberInt('50000000000')})
4959
4960 // but if the data stored as a 64 bit double then it works
4961 > db.companies.insertOne({valuation: 50000000000})
4962
4963 > db.companies.find()
4964 { "_id" : ObjectId("5f3e81e7d0209e4d3a0ec072"), "valuation" : -1539607552 }
4965 { "_id" : ObjectId("5f3e8635d0209e4d3a0ec073"), "valuation" : 50000000000 }
4966 { "_id" : ObjectId("5f3e866cd0209e4d3a0ec074"), "valuation" : 705032704 }
4967
4968 // to store biggest possible number as Integers
4969 // always have to use quotation marks
4970 // basically it works as a string
4971 > db.companies.insertOne({valuation: NumberLong('50000000000')})
4972 > db.companies.find()
4973 { "_id" : ObjectId("5f3e81e7d0209e4d3a0ec072"), "valuation" : -1539607552 }
4974 { "_id" : ObjectId("5f3e8635d0209e4d3a0ec073"), "valuation" : 50000000000 }
4975 { "_id" : ObjectId("5f3e866cd0209e4d3a0ec074"), "valuation" : 705032704 }
4976 { "_id" : ObjectId("5f3e86ecd0209e4d3a0ec075"), "valuation" :
NumberLong("50000000000") }
4977
4978 -----Doing Maths with Floats Int32 Int64-----
4979
4980 > db.accounts.insertOne(name: 'Max', amount: '34234253458373534574524524')
4981 // add a small number
4982 > db.accounts.insertOne(name: 'Max', amount: '10')
4983
4984 // $inc or any math calculation does not work with string value

```

```

4985 > db.accounts.updateOne({}, {$inc: {amount: 10}})
4986
4987 // have to insert a Integers value
4988 > db.accounts.deleteMany()
4989 > db.accounts.insertOne(name: 'Max', amount: NumberInt('10'))
4990
4991 // here 10 as a double value mongoDB convert the sum as double
4992 > db.accounts.updateOne({}, {$inc: {amount: 10}})
4993
4994 // if update the number with wrapping with NumberInt then the final output be a int
4995 > db.accounts.updateOne({}, {$inc: {amount: NumberInt('10')}})
4996
4997 // let delete the document
4998
4999 > db.companies.deleteMany({})
5000
5001 // insert a large number
5002 > db.companies.insertOne({valuation: NumberLong('34234253458373534574524524')})
5003
5004 // to calculate math operation with the large number NumberLong should be include
5005 // in that number
5006 // this is incorrect
5007 > db.companies.updateOne({}, {$inc: {valuation: 1}})
5008 // this is correct
5009 > db.companies.updateOne({}, {$inc: {valuation: NumberLong('1')}})
5010
5011 -----What's Wrong With Normal Doubles-----
5012
5013 > db.science.insertOne({a: 0.3, b: 0.1})
5014
5015 > db.science.findOne()
5016
5017 // let execute maths calculation
5018 > db.science.aggregate([{$project: {result: {$subtract: ['$a', '$b']}}}])
5019
5020 // here should be the subtract value is 0.2
5021 // but it's come
5022 { "_id" : ObjectId("5f3e9ec9d0209e4d3a0ec079"), "result" : 0.19999999999999998 }
5023
5024 // so have to fix the issue
5025
5026 -----Working With Decimal 128bit-----
5027
5028 // to get the exact subtract value, have to use NumberDecimal constructor
5029
5030 > db.companies.deleteMany({})
5031 > db.science.insertOne({a: NumberDecimal("0.3"), b: NumberDecimal("0.1")})
5032
5033 > db.science.find().pretty()
5034 {
5035   "_id" : ObjectId("5f3ea3ced0209e4d3a0ec07b"),
5036   "a" : NumberDecimal("0.3"),
5037   "b" : NumberDecimal("0.1")
5038 }
5039
5040 // now getting the exact value
5041 > db.science.aggregate([{$project: {result: {$subtract: ['$a', '$b']}}}])
5042 { "_id" : ObjectId("5f3ea3ced0209e4d3a0ec07b"), "result" : NumberDecimal("0.2") }
5043
5044 // let execute another query
5045 > db.science.updateOne({}, {$inc: {a: 0.1}})
5046
5047 > db.science.updateOne({}, {$inc: {a: 0.1}})
5048 { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
5049 > db.science.find().pretty()
5050 {
5051   "_id" : ObjectId("5f3ea3ced0209e4d3a0ec07b"),
5052   "a" : NumberDecimal("0.4000000000000000"),
5053   "b" : NumberDecimal("0.1")

```

```

5053     }
5054
5055     // so to get the right value
5056     > db.science.updateOne({}, {$inc: {a: NumberDecimal("0.1")}})
5057     { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
5058     > db.science.find().pretty()
5059     {
5060       "_id" : ObjectId("5f3ea3ced0209e4d3a0ec07b"),
5061       "a" : NumberDecimal("0.5000000000000000"),
5062       "b" : NumberDecimal("0.1")
5063     }
5064
5065     // NumberDecimal means getting high precision decimal
5066
5067     > db.number.insertOne({num: 0.1})
5068     > db.number.stats()
5069     "size" : 33
5070
5071     > db.number.deleteMany({})
5072
5073     > db.number.insertOne({num: NumberDecimal("0.1")})
5074     > db.number.stats()
5075     "size" : 41
5076
5077
5078
5079 -----
5080 16-transactions
5081 -----
5082
5083 -----Transactions-----
5084
5085             User deletes Accounts
5086   Users Collection           Posts Collection
5087 -----
5088   | { User Document } -----> { Post Document } |
5089   |                               \                |
5090   |           related             \                |
5091   |                               \-----> { Post Document } |
5092   | Should be deleted together |
5093 -----
5094
5095     // always have to change ip address into mongoDB cloud cluster
5096
5097     // first have access the mongoDB cloud
5098     mongo "mongodb+srv://mytestingcluster.n7v1t.mongodb.net/<test1>" --username
5099     mijanur
5100
5101     > use blog
5102     > db.users.insertOne({name: 'Max'})
5103     > db.posts.insertMany([{title: 'A js post', views: 23, userId:
5104       ObjectId("5f4163d6526c4846e4c6felb")}, {title: 'Group discussion', views: 2,
5105       userId: ObjectId("5f4163d6526c4846e4c6felb")}])
5106
5107     // have to execute the mongo session to work with the Transactions
5108
5109     > const session = db.getMongo().startSession()
5110     > session.startTransaction()
5111
5112     > const usersCol = session.getDatabase('blog').users
5113     > const postsCol = session.getDatabase('blog').posts
5114
5115     // this is basically remove from session
5116     > usersCol.deleteOne({_id: ObjectId("5f4163d6526c4846e4c6felb")})
5117
5118     // this command also successfully execute(this comes from cloud not session), but
5119     we deleted the user before

```

```
5116 > postsCol.deleteMany({userId: ObjectId("5f4163d6526c4846e4c6fe1b")})
5117
5118 MongoDB Enterprise atlas-9fuf07-shard-0:PRIMARY> usersCol.deleteOne({_id:
5119 ObjectId("5f4163d6526c4846e4c6fe1b")})
5120 { "acknowledged" : true, "deletedCount" : 1 }
5121
5121 // it basically deleted from cache but not from real server
5122 > db.users.find().pretty()
5123 { "_id" : ObjectId("5f4163d6526c4846e4c6fe1b"), "name" : "Max" }
5124
5125 // to execute fully delete from cloud have to commit Transactions
5126 > session.commitTransaction()
5127
5128 // now deleted from cloud
5129 > db.users.find().pretty()
5130
5131 // can also abort --> all things are trying to fully delete
5132
5133 // so these actions either succeed together or they fail together. That is the idea
5134 behind the transactions.
5135
5136 // this is basically comes from atomicity
5137 // get an atomicity in operation level not just on a document level
5138 // so need cross operation consistency
5139
5140 -----
5141 -----
5142 -----
5143
5144
```