**MongoDB Cheat Sheet**

1. **Show all databases are in the server:**

show dbs

1. **Create or use existing database:**

use flights

1. **Insert Query**
   1. **Insert One** Document

db.flightData.insertOne({departureAirport: “MUC”, arrivalAirport: “SFO”, aircraft: “Airbus A380”, distance: 12000, intercontinential: true})

* 1. **Insert Many** Document – As a 2nd parameter pass the *ordered* like below. By default, its value is true that means when inserting documents, if one documents are failed, then next documents can’t go ahead to store into db.

db.flightData.insertMany([{1st document}, {2nd document}, {3rd document}], ordered: false)

* 1. **Insert –** Can insert single or multiple documents though it’s not recommended.

db.persons.insert([{document1}, {document2}])

* 1. **writeConcern -**bydefault, w = 1, otherwise 0. And j for jurnal, bydefault undefined or false, otherwise true. And also can define wtimeout.

db.persons.insertOne({…document…}, {writeConcert: {w: 0, j: true, wtimeout: 200}})

* 1. MongoDB CRUD Operations are Atomic on the Document Level including Embedded Documents.
  2. Import from files

mongoimport tv-shows.json -d movieData -c movies –jsonArray --drop

1. **Read –** Only available **Query Selectors** & **Projection Operators**
   1. Read all data

db.flightData.find().pretty()

* 1. Read with specific thing

db.flightData.find({distance: 12000}).pretty()

* 1. Read with Greater Than Operator (Range Filter)

db.flightData.find({distance: {$gt: 900}})

* 1. Only fetch 1st citerial that mate with condition

db.flightData.findOne({distance: {$gt: 900}})

**c. .find()** don’t return all documents together, it returns 20 documents and also return with is a cursor that helps to access the next documents. To fetch all documents 🡪

db.flightData.find().toArray()

1. Like **.toArray()** another method

db.passengers.find().forEach((passengerData => {printjson(passengerData)}))

|  |  |  |
| --- | --- | --- |
| Query Selectors | | Projection Operators |
| Comparison | Evaluation | $ |
| Logical | Array | $elemMatch |
| Element | Comments | $meta |
| Geospatial | | $slice |

1. **Official Document Resource** <https://docs.mongodb.com/manual/reference/operator/query/>
2. **Read embedded documents**

db.movies.find({“rating.average”: {$gt: 7}}).pretty()

1. **Read Array nested documents**
   1. Find that document where inside the array, one element matches

db.movies.find({genres: “Drama”}).pretty()

* 1. Find exactly that array

db.movies.find({genres: [“Drama”]}).pretty()

1. **$in**: Only read data which runtime is 30 or 42.

db.movies.find({runtime: {**$in**: [30, 42]}}).pretty()

1. **$nin**: Only read that datas where thease runtime is not there.

db.movies.find({runtime: {**$nin**: [30, 42]}}).pretty()

1. **$or** query and **$nor** query

db.movies.find({**$or**: [{“rating.average”: {**$lt**: 5}}, {“rating.average”: {$gt: 9.3}}]}).pretty()

db.movies.find({$nor: [{“rating.average”: {**$lt**: 5}}, {“rating.average”: {$gt: 9.3}}]}).pretty()

1. **$and** query

db.movies.find({**$and**: [{“rating.average”: {**$gt**: 9}}, {genres: “Drama”}]}).pretty()

db.movies.find({“rating.average”: {**$gt**: 9}, genres: “Drama”}).pretty()

1. **$not** query

db.movies.find({runtime: {**$not**: {$eq: 60}}}).count()

**$ne:** Alternative query of above query.

db.movies.find({runtime: {$ne: 60}}).count()

1. **$exists**: Find the documents that have age key

db.users.find({age: {**$exists**: true, **$gte**: 30}}).pretty()

exists and also value is null.

db.users.find({age: {**$exists**: true, **$ne**: null}}).pretty()

1. **$type:** Find from db with datatypes

db.users.find({phone: {**$type**: “number”}}).pretty()

Can also find with multiple types like

db.users.find({phone: {**$type**: [“double”, “string”]}}).pretty()

1. **$regex:** Find documents where some word is in used in summary key.

db.movies.finds({summary: {**$regex**: /musical/}}).pretty()

1. **$expr:** Find that document where volume key value is greater than target key value

db.sales.find({**$expr**: {**$gt**: [“**$**volume”, “**$**target”]}}).pretty()

db.sales.find({**$expr**: {**$gt**: [{**$cond**: {if: {**$gte**: [“**$**volume”, 190]}, then: {**$subtract**: [“**$**volume”, 10]}, else: “**$**volume”}}, “**$**target”]}}).pretty()

1. **Embedded documents inside array –** Inside *hobbies* array, every elements are embedded documents and every embedded documents have two key-value pairs, *title* and *frequency*.

db.users.find({“hobbies.title”: “Sports”}).pretty()

1. **$size –** Find that documents where hobbies key has an array and inside the array, there are 3 elements.

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

1. **$all-** Find that documents which has key genre and values are inside an array and that array has the two elements. It doesn’t maintain the order also.

db.moviestarts.find({genre: {$all: [“action”, “thriller”]}}).pretty()

1. **$elemMatch –** Every document has habbies key that holds an array of embedded documents in which 1st element is hobbies title and 2nd element is frequency. In first query, does not check both title and frequencies in same embedded documents or not. It checks through the whole array. To full fill this, $elemMatch takes place in this senario.

db.users.find({$and: [{“hobbies.title”: “Sports”}, {“hobbies.frequency”: {$gte: 2}}]}).pretty()

 db.users.find({hobbies: {$elemMatch: {title: “Sports”, frequency: {gt: 3}}}}).count()

1. **Cursor –** Through cursor

const dataCursor = db.movies.find().next()

dataCursor.forEach(doc => {printjson(doc)})

**Check whether next document is there or not**

dataCursor.hasNext()

1. **Sorting Cursor Results:–** 1 means ascending, -1 means descending

db.movies.find().sort({“rating.average”: -1, runtime: 1}).pretty()

1. **Skipping & Limiting Cursor Result:** In this case, order is important. Sort 🡪 skip 🡪 limit

db.movies.find().sort({condition}).**skip(10).limit(10)**.pretty()

1. **Projection –** 1 means included and 0 means excluded and that is by-default. Don’t hant to pass by telling 0.

db.movies.find({}, {name: 1, genres: 1, “rating.average”: 1, image: 0}).pretty()

  If any key has array values, then projection -

db.movies.find({genres: “Drama”}, {“genres.$”: 1}).pretty()

1. **$slice –** genres have more array elements, but want only 2

db.movies.find({“rating.average”: {$gt: 9}}, {genres: {**$slice**: 2}}).pretty()

db.movies.find({“rating.average”: {$gt: 9}}, {genres: {**$slice**: [1, 2]}}).pretty()

Last query, skip the 1st element and project nesxt 2 elements

1. **Update**
   1. **Update One** with **$set** Document

db.flightData.**updateOne**({distance: 12000}, {**$set**: {marker: “delete”}})

* 1. **Update Many** Document

db.flightData.**updateMany**({selecting area}, {**$set**: {marker: “toDelete”}})

* 1. **Update** – This will override the old data.

db.flightData.**update**({\_id: ObjectId(“something”)}, {delayed: false})

* 1. **replaceOne**

db.flightData.**replaceOne**({\_id: ObjectId(“something”)}, {…document values})

* 1. **Increment ($inc):** Increment 1 of the age.

db.users.updateOne({name: “Manuel”}, {$inc: {age: 1}, $set: {another update}})

**Decrement($inc):** Decrement same as increment by passing negative value.

db.users.updateOne({name: “Manuel”}, {$inc: {age: -1}, $set: {another update}})

* 1. **$min**- It’s update only when new value is lower than the existing value.

db.users.updateOne({name: “Chris”}, {$min: {age: 35}})

**$max** – Same as $min when new value is hogher than the existing value, then only update.

**$mul** – Multiply with existing value and then new value is stored.

db.users.updateOne({name: “Chris”}, {$mul: {age: 1.1}})

* 1. **$unset**: Drop the field mean delete key-value pairs.

db.users.updateMany({isSporty: true}, {$unset: {phone: “”}})

* 1. **Renaming Fields($rename):**

db.users.updateMany({}, {$rename: {age: “totalAge”}})

* 1. **upsert():** If the document doesn’t exists then will be created otherwise will update. Bydefault it’s value is false.It have to pass as a 3rd argument.

db.users.updateOne({name: “Maria”}, {$set: {age: 29, hobbies: [{title: “Good food”, frequency: 3}, isSporty: true]}}, {upsert: true})

* 1. **Update Array Element-** See read elemMatch for details. And updating. **.$** is used to update that selected array embedded documents, not to override.

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

* 1. **All Array Elements-**

db.users.updateMany({totalAge: {$gt: 30}}, {$inc: {“hobbies**.$[].**frequency”: -1}})

* 1. **Finding & Updating Specific Fields:**

db.users.updateMany({“hobbies.frequency”: {$gt: 2}}, {$set: {“hobbies**.$[el].**goodFrequency”: true}}, **{upsert: true, arrayFilters: [{“el.frequency”: {$gt: 2}}**]})

* 1. **Adding Elements to Array:**
     1. **For One Embedded Document**

db.users.updateOne({name: “Maria”}, {$push: {hobbies: {title: “Sports”, frequency: 2}}}})

* + 1. **For More Than One Embedded Document**

db.users.updateOne({name: “Maria”}, {$push: {hobbies: {$each: [{title: “Good Wine”, frequency: 1}, {title: “Hiking”, frequency: 2}], $sort: {frequency: -1}, $slice: 1}}})

* 1. **Removing Elements from Array:**
     1. **Just Remove (No Citerial)**

db.users.updateOne({name: “Maria”}, {**$pull**: {hobbies: {title: “Hiking”}}})

* + 1. **Last Element Remove:**

db.users.updateOne({name: “Chris”}, {**$pop**: {hobbies: 1}})

* + 1. **First Element Remove:**

db.users.updateOne({name: “Chris”}, {**$pop**: {hobbies: -1}})

* 1. **$addToSet:**Work like push but only works with unique value. If element is already exsits, push will work. But this does not work.

db.users.updateOne({name: “Maria”}, **{$addToSet**: {hobbies: {title: “Hiking”, frequency: 2}}})

1. **Delete**
   1. **Delete One Document**

db.flightData.deleteOne({departureAirport: “TXL”})

* 1. **Delete Many Document**

db.flightData.deleteMany({marker: “toDelete”})

1. **Projection:** 1 for included and 0 for excluded

db.passengers.find({}, {name: 1, \_id: 0}).pretty()

1. **Embedded Document:** Inside a document, can have multiple child documents. In mongodb, this child documents’ depth will be max 100 and must max memory within 16mb. Can also possible to use array without using embedded document.

**Find embedded data**:

db.flightData.find({“status.description”: “on-time”}).pretty()

**Find array data:**

db.passengers.find({hobbies: “sports”}).pretty()

db.passengers.findOne({name: “Albert Twostone”}).hobbies

1. **db.stats()** -> Tells the summary of a database

db.stats()

1. Generally, in shell, use **64INT Number** to store numbers. If want to store **32INT number**🡪

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

1. **Drop database**

db.dropDatabase()

1. **lookup() Operator –** Merging data of different collections in one step

db.books.aggregate([$lookup: {from: “authors”, localField: “authors”, foreignField: “\_id”, as: “creators”}])

1. **Schema Validation -** Create Collection with conditions

db.createCollection('posts', {

  validator: {

    $jsonSchema: {

      bsonType: 'object',

      required: ['title', 'text', 'creator', 'comments'],

      properties: {

        title: {

          bsonType: 'string',

          description: 'must be a string and is required',

        },

        text: {

          bsonType: 'string',

          description: 'must be a string and is required',

        },

        creator: {

          bsonType: 'objectId',

          description: 'must be a objectId and is required',

        },

        comments: {

          bsonType: 'array',

          description: 'must be an array and is required',

          items: {

            bsonType: 'object',

            required: ['text', 'author'],

            properties: {

              text: {

                bsonType: 'string',

                description: 'must be a string and is required',

              },

              author: {

                bsonType: 'objectId',

                description: 'must be a objectId and is required',

              },

            },

          },

        },

      },

    },

  },

});

1. **Change schema validation:**

db.runCommand({

  collMod: 'posts',

  validator: {

    $jsonSchema: {

      bsonType: 'object',

      required: ['title', 'text', 'creator', 'comments'],

      properties: {

        title: {

          bsonType: 'string',

          description: 'must be a string and is required',

        },

        text: {

          bsonType: 'string',

          description: 'must be a string and is required',

        },

        creator: {

          bsonType: 'objectId',

          description: 'must be a objectId and is required',

        },

        comments: {

          bsonType: 'array',

          description: 'must be an array and is required',

          items: {

            bsonType: 'object',

            required: ['text', 'author'],

            properties: {

              text: {

                bsonType: 'string',

                description: 'must be a string and is required',

              },

              author: {

                bsonType: 'objectId',

                description: 'must be a objectId and is required',

              },

            },

          },

        },

      },

    },

  },

  validationAction: 'warn',

});

1. **How does mongodb do:** As an argument can pass, **“queryPlanner”** (Show Summary for executed query + winning plan) or **“executionStats”** (Show detailed summary for executed query + winning plan + possibly rejected plans) or **“allPlansExecution” (**Show detailed summary for executed query + winning plan + winning plan decision process)

db.contacts.explain().find()

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

1. **Indexing:** 
   1. **A Single Field Index:** 1 for ascending and -1 for descending

db.contacts.createIndex({“dob.age”: 1})

* 1. **Drop the single field index:**

db.contacts.dropIndex({“dob.age”: 1})

* 1. **Compound Indexes:**

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

* 1. It also helps when sorting. Without indexing, when sording, mondodb has thresold 32mb. If it crosses then sorting is not possible. In that case, indexing helps it.
  2. **View indexes –**

db.contacts.getIndexes()

* 1. **Unique Indexes –** In this index, email will be unique, can’t any values that are dublicates.

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

Sometimes, someone has no email, but in that case, because of only telling unique, mongodb gives error. For this solution, can store without email and also every email will be unique -

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

* 1. **Partial Filters-**

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

* 1. **Time-To-Live(TTL) Index:** Session data or anything like that which are self-destroy data with time.It’s only available only on **ISODates.** After that time, the data will be deleted from db. And it is only available on single field index.

db.sessions.createIndex({createdAt: 1}, {expireAfterSeconds: 10})

* 1. **Multi-Key Indexes:** It works on array like documents. It pulls out the values and create indexes.It behaves like normal indexes but works differently inside.

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

* 1. **Text Indexes:** While indexing, removes all stopwords and store keywords.

db.products.createIndex({description: “text”})

**Search –** Text Indexes only available only one parameter, in this case, description. Because, it’s costly of performance.

db.products.find({$text: {$search: “awesome”}})

 db.products.find({$text: {$search: “\”awsome book\””}}).pretty()

 MondoDB defines score automatically means it gives piority. If one document has only awesome and another document has both key words. Then mongodb gives high score to that documents where direct search is found. It’s automatically sorted. Though can sort by passing .sort()

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

**Drop Text Indexes** – Little bit header to drop text indexes, only possible by passing index name.

db.products.dropIndex()description\_text

**Combined Text Indexes** – For text indexes, only possible for only one field. But if required multiple fields, then can create text indexes by combined them together.

db.products.createIndex({title: “text”, description: “text”})

* 1. **Default Language & Using Weight-**Weight is that when searching some text using indexing, which part will be piority more.

db.products.createIndex({title: “text”, description: “text”}, {default\_language: “english”, weights: {title: 1, description: 10}})

 Can also turn on the **$caseSentivity**. By-default, false.

db.products.find({$text: {$search: “”, $caseSensitive: true}})

* 1. **Background Indexes:** Till now, creating indexes are foreground indexes. When creating this type indexing, can’t do inserting or changing or onther else with the db. This will be bad for production purpose. So background indexing plays this role. That is the one difference. Another difference between then is that foregroung indexes are faster than background indexes**.**

db.ratings.createIndex({age: 1}, {background: true})

1. **Geospecial Data-** 
   1. **Adding GeoJSON Data:** 1st element should belongitude and 2nd value should be latitude.

db.places.insertOne({name: "California Academy of Science", location: {type

: "Point", coordinates: [-122.4724356, 37.7672544]}})

* 1. **Find near locations-** Find nearer locations that stored into the database. For this, 1st have to create index, otherwise will be failed by passing **2dspehere** parameter.

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

 Then search the nearer locations. Can also pass the distance required to be nearer in **meter**. location keyword is not researved, **type** and **coordinates** these two words are reserved.

db.places.find({location: {$near: {$geometry: {type: "Point", coordinates:

[-122.471114, 37.771104]}, $maxDistance: 500, $minDistance: 10}}}).pretty()

* 1. **Find all places inside the certain area:**

const p1 = [-122.4547, 37.77473]

const p2 = [-122.45303, 37.76641]

const p3 = [-122.51026, 37.76411]

const p4 = [-122.51088, 37.77131]

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

* 1. Finding Out If a User is inside a specific area:

db.areas.find({area: {$geoIntersects: {$geometry: {type: "Point", coordinates: [-122.49089, 37.76992]}}}}).pretty()

* 1. **Finding Places Within a Certain Radius:**

db.places.find({location: {$geoWithin: {$centerSphere: [[-122.46203, 37.77286], 1 / 6378.1]}}}).pretty()

1. **Aggregation Framework-**
   1. **Example-1**

db.persons.aggregate([

  { $match: {gender: 'female'} },

  { $group: {\_id: { state: "$location.state" }, totalPersons: { $sum: 1 } } },

  { $sort: { totalPersons: -1 } }

]).pretty();



1. **Numeric Data-**

|  |  |  |  |
| --- | --- | --- | --- |
| **Integers (int32)** | **Longs (int62)** | **Doubles (64bit)** | **“High Precision Doubles” (128bit)** |
| Only full Numbers | Only fill Numbers | Numbers with Decimal Places | Numbers with Decimal Places |
| -2147483648 to 2147483647 | -9223372063854775808 to 9223373036854775807 | Decimal values are approximated | Decimal values are stored with high precision (34 decimal digits) |
| Use for normal integers | Use for large integers | Use for floats where high precision is not required | Use for floats where high precision is required |

1. **Int32:**

db.persons.insertOne({age: NumberInt(“29”)})

1. **Int64:**

db.companies.insertOne({valuation: NumberLong(“2147483648”)})

1. Normal Doubles:

db.science.insertOne({a: 0.3, b: 0.1})

1. Decimal 128bit:

db.science.insertOne({a: NumberDecimal(“0.3”), b: NumberDecimal(“0.1”)})