db.col.find({name:”Arun”}).count()

db.col.find({name:”Arun”, location:”US”}).count() // and conditions

db.col.find({“tomato.meter”:100}); // for nested documents.

**Arrays**

db.col.find({“writers”:[“Arun gopan”, “Nanda kumar”]}).count(); // looks for exact match, order matters

db.col.find({“name”:”Arun”}).count() -- looks for values Arun inside the array

db.col.find({“name.0”:”Arun”}).count() -- looks for values Arun inside the array in the first position

**Cursors and projections**

var c = db.details.find();

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

c.objsLeftInBatch(); // returns the number of documents left to read

You can just call doc() to iterate through

Use projection to specify the data retrieved my mongodb

db.details.find({“name”:”Arun”}, {title:1}).pretty();

val 1 : return, 0: ignore the fields

**Comparison operators**

* $gt / $lt: greater than / less than
  + db.details.find({ “runtime”: {$gt:90, $lt:120}});
* $gte / $lte: greater than or equal to / less than or equal to
  + db.details.find({ “runtime”: {$gte:90, $lte:120}});
* db.details.find({  
   “tomato.meter”: {$gte : 90},  
   runtime: {$gt : 180}  
  });
* $ne : not equal to / or no field at all
  + db.details.find({rated: {$ne:”unrated”}});
* $in : which contains the given value
  + db.details.find({rated: {$in: [“G”, “PG”]}})

**Element operators**

* $exists : checks whether the field exists and return the set of document which holds it.
  + db.details.find({“tomato.meter”:{$exists:true}});
  + true : return the documents which has the field
  + false : returns the documents which doesn't has that field
* $type: compares with the type of a field.
  + db.details.find({“\_id”: {$type: “string”}});

**Logical operators**

* $or: performs an or operation
  + db.details.find( {  
     $or : [  
     {“tomato.meter”: { $gt: 95} },  
     {“metacritic”: { $gt: 88} }  
     ]  
    } ).count();
* $and : perform an and operation
  + db.details.find( {  
     $and : [  
     {“rating”: { $ne: null} },  
     {“rating”: { $exists: true} }  
     ]  
    } ).count();
  + Used when you want to specify multiple condition for the same field, if you do it the usual way, the values in the json object will get overwritten because of the same value for the key element.

**Array operators**

* $all : return documents which has all the values passed to it.
  + db.details.find( {  
     genres : {  
     $all : [“Comedy”, “Crime”, “Drama”]  
     }  
    })
* $size : match documents based on the length of array
  + db.details.find({ countries: { $size : 1}})
* $elemMatch :
  + db.details.find({  
     boxOffice: {  
     $elemMatch : {  
     country : “UK”,   
     revenue : { $gt : 15 }  
     }  
     }  
    })
  + Sample document could be something show below.
  + boxOffice : [  
     { “country” : “India”, “revenue”:41 },  
     { “country” : “UK”, “revenue”:41 },  
     { “country” : “US”, “revenue”:41 },  
     { “country” : “Canada”, “revenue”:41 },  
    ]
  + In the above query it will check documents satisfying both the conditions together.
  + db.users.find(  
     {  
     "grades":   
     {  
     $elemMatch: {  
     $or:[  
     {"topic":"Math"},  
     {"marks":100}  
     ]  
     }  
     }  
     }  
    ).pretty();

**Updating documents**

* updateOne
  + db.details.updateOne(  
     {“title”:”The Martian”},  
     {$set : {  
     Poster:”test content for posted”  
     }}  
    );
  + First parameter for the updateOne is the filter criteria
  + Second is the fields which we want to update
    - It will update poster property of all the documents to the new value
    - If not present it will insert a new property poster to all the documents.
* **$inc :** increments the value of a field
  + db.details.updateOne({title:”The Martian”},  
     {  
     $inc : {“tomato.reviews”:3, ”tomato.userReviews”: 25}  
     }  
    );
  + The above query will update the tomato.reviews field by updating the current value by 3 and tomato.userReviews by 25;
* **Update array function** // to be checked  
  + **$push** : add documents into an array
    - db.details.updateOne ({title:”The Martian”},  
       {$push:{ reviews : {  
       rating : 4.5,  
       reviewer : ‘Arun’,  
       reviewText : “This is a sample review”  
       }  
       }}  
      );
  + **$each** : used along with $push, which create each item as separate documents in the array. If you don't use $each the entire array we pass to $push will be added as a single element in the array.  
    - db.training.updateOne({title:"Judjment day"},  
       { $push: { reviews:{   
       $each: [  
       {rating:4.5, reviewer:"James", "reviewtext":"This is a sample review"},  
       {rating:4.5, reviewer:"James", "reviewtext":"This is a sample review"}  
       ]  
       }  
       }  
      });
    - **without $each**
    - {  
       "\_id" : ObjectId("5863e0397aed0d83576cdf51"),  
       "title" : "Avatar",  
       "year" : "2010",  
       "reviews" : [  
       [  
       {  
       "rating" : 4.5,  
       "reviewer" : "James",  
       "reviewtext" : "This is a sample review"  
       },  
       {  
       "rating" : 4.5,  
       "reviewer" : "James",  
       "reviewtext" : "This is a sample review"  
       }  
       ]  
       ]  
      }
    - **with $each**
    - {  
       "\_id" : ObjectId("5863e0417aed0d83576cdf52"),  
       "title" : "Judjment day",  
       "year" : "1990",  
       "reviews" : [  
       {  
       "rating" : 4.5,  
       "reviewer" : "James",  
       "reviewtext" : "This is a sample review"  
       },  
       {  
       "rating" : 4.5,  
       "reviewer" : "James",  
       "reviewtext" : "This is a sample review"  
       }  
       ]  
      }
  + **$slice & $position**
    - **$slice:** to limit the number of records to be entered in the array
    - **$position :** which position to insert the new element after exceeding the size.
  + db.mov ies.updateOne({title:"Judjment day"},  
     { $push: { reviews: {  
     $each: [  
     {rating:7.5, reviewer:"James", "reviewtext":"This is a sample review"}],   
     $position:0,   
     $slice:6  
     }  
    }});
  + **$unset :** remove property from a document
    - db.details.updateMany({rated:null},   
       { $unset : {rated: “”}}  
      );
    - db.details.find({rated:null}); // returns the documents that don't have a rated field.
  + **$upsert:** while inserting if the document is not found it will create a new document if upsert value is true;
    - db.movies.updateOne({title:"abc"},   
       {$set:{title:'ABC', year:'2000'}},   
       {upsert:true}  
      );
* **replaceOne :** 
  + db.movies.replaceOne({“imdb”: detail.imdb.id}, detail);
  + detail is the variable which holds the entire set of documents.

**MongoDB Schema Design**

While designing schema we need to think about the application data patterns, like what pieces of data are used together, readonly data etc. Then we will organize our data within MongoDB to suit the application data access patterns.

**Storage Engine**

Storage engine is the interface between persistent storage (disk) and the database itself (mongodb). Mongodb talks to disk through storage engine.

Your script (nodejs, php, pythons) => mongodb => storage engine => disks.

Types of storage engines:

* MMAPv1=> default
  + Supports collection level locking
  + In place updates
  + Power of two sizes
* Wired tiger
  + Document level concurrency
  + Compression for data and indexes (automatically compress when files moved into disk and in memory it maintains uncompressed data, which might otherwise affect the performance of access of data from memory)
  + Manages memory to access file
  + No inplace updates, which means it will mark the file which is being updated and allocates different memory for it and reclaims the unused space.
  + **killall mongodb => kills all mongodb process**
  + **Start mongodb with Wired Tiger**
    - Create a directory WT (any name will do)
    - mongod -dbpath WT -storageEngine wiredTiger

**Indexes**

* **Creating indexes**
  + db.students.createIndex({“student\_id” : 1});
  + findOne() is much faster than find(), reason is findOne() will stop searching the collections once it find a matched document where as the find() the find() will still keep searching all the records in the collections
* **Explain command:**  To check the index.
  + db.students.explain().find({“student\_id”: 1});
  + explain(true) will get the documents examined information
* **Discovering indexes**
  + **Retrieve all the index**
    - db.students.getIndexes();
  + **Delete an index**
    - db.companies.dropIndex({"name":1});

**Multikey Indexes**

* Not allowed to specify multi key index if both are array values.
* **$elemMatch :** is the prefered methods for matching elements using multiple conditions.

**Unique indexes**

db.fruits.createIndex({"name":1}, {unique:true});

**Index creation, Sparse**

Sparse index : which are indexes that can be used when the index key is missing from some of the documents.

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

Create indexes

* Foreground
  + Default in mongodb
  + Faster than background
  + Blocks all writers and reader within in the same database
* Background
  + Relatively slower
  + Don't block readers and writers
  + Only index creation one at a time
  + db.employees.createIndex({'cell': 1},{ background:true})

**Explain**

* Cannot explain an insert command
* db.employee.explain().find();

**Covered queries**

Is a query where the query itself can be satisfied entirely with an index and hence “0” documents needs to be inspected to satisfy the query and will lot faster.

If we run a query and fetch values where index is defined, thats what a covered query is.

**Geospatial indexes** (support querying x/y coordinate values)

db.employess.ensureIndex({location:’2d’, type:1}); // location is the column name, it could be with any field, [x, y]

db.employee.find({“location”: {$near : [50,50]}});

**Geospatial spherical**

**2dsphere** : Index documents which has latitude and longitude data in them.

Mongodb makes use of **GeoJSON** format for these geographic data structures.

{

"\_id" : ObjectId("58725486ae3a4e652237ab10"),

"name" : "Apple Store",

"city" : "Palo Alto",

"location" : **{**

**"type" : "Point",**

**"coordinates" : [**

**-122.1691291,**

**37.4434854**

**]**

**}**,

"type" : "Retail"

}

In the above document whatever highlighted in bold is a GeoJSON document.

**“type”, “Point”, “coordinates” are keywords.**

db.places.find( {location:  
 {$near:

{$geometry:

{

type:"Point",

coordinates:[-122.166641, 37.4278925]

},

$maxDistance:2000

}

}

}).pretty();

Using **$geometry,** you specify the GeoJSON

**Fulltext search index**

db.companies.ensureIndex({"overview":"text"});

db.companies.find({$text:  
 {

$search:"property"

}

}).count();

db.companies.find(

**{**

**$text:**

**{ $search:"demand video player" }**

**}, // performing text search**

{

overview:1,

**score:**

**{**

**$meta:'textScore'**

**}**

**}) // creating a score field for the textscore**

.**sort(**

**{**

**score:**

**{**

**$meta:'textScore'**

**}**

**}); // sorting documents based on textScore**

**Designing and using indexes**

Goal : efficient read / write operations

Selectivity is the primary factor that determines how efficiently an index can be used.

Ideally, the index enables us to select only those records required to complete the result set, without the need to scan a substantially larger number of index keys (or documents) in order to complete the query.

Selectivity determines how many records any subsequent operations must work with. Fewer records means less execution time.

**Logging slow queries**

Mongodb automatically log slow queries above 100ms

**Profiling**

Writes entries to **system.profile** for any slow queries. There are 3 levels to the profiler.

* Level 0
  + Default level => off
* Level 1
  + Log slow queries
* Level 2
  + Log all queries

**Mongotop**

Used to check all the process along with the time spent for both read and write operations

**Mongostat**

Is a performance tuning command, gives information about the reades, writes and various operations which happened in each second.

**Sharding**

Technique for splitting up a large collection among multiple servers.

Application => mongos => different shard servers (mongod)

Each shard can have multiple replica sets.

**mapReduce**

db.users.mapReduce(

function() {

for(num in this.grades) {

emit(this.grades[num].topic, this.grades[num].marks)

}

},

function(key, values) {

return Array.sum(values);

},

{

query:{"grades.topic":"Math"},

out:"marks\_total"

}

).find();

Collection would look like

[

{

"firstname":"Arungopan",

"lastname":"Gopakumar",

"email":"arun@gmail.com",

"interest":["Bikes", "Cricket", "Movies", "Music"],

"address":{

"communication":"United States",

"permanent":"India"

},

"grades":[

{"topic":"Math", "marks":70},

{"topic":"Computers", "marks":90},

{"topic":"Physics", "marks":80},

{"topic":"Graphics", "marks":60}

]

},

{

"firstname":"Nandakumar",

"lastname":"Purohit",

"email":"nanda@gmail.com",

"interest":["Movies", "Music"],

"address":{

"communication":"Edison",

"permanent":"Banglore"

},

"grades":[

{"topic":"Math", "marks":100},

{"topic":"Computers", "marks":80},

{"topic":"Physics", "marks":90},

{"topic":"Graphics", "marks":70}

]

},

{

"firstname":"Naveen",

"lastname":"Suresh",

"email":"naveen@gmail.com",

"interest":["Bikes", "Cricket", "Movies"],

"address":{

"communication":"NJ",

"permanent":"Kerala"

},

"grades":[

{"topic":"Math", "marks":90},

{"topic":"Computers", "marks":100},

{"topic":"Physics", "marks":70},

{"topic":"Graphics", "marks":80}

]

}

]