CS572 Modern Web Applications Programming
Maharishi University of Management
Department of Computer Science
Assistant Professor Asaad Saad

PREV NEXT

OUTLINE

Search...

CRUD in MongoDB

There is no special SQL language to perform CRUD in MongoDB.

Many CRUD operations exist as methods/functions on objects in programming language API, NOT as separate language.

CRUD	MongoDB	SQL
Read	find()	select
Create	insert()	insert
Update	update()	update
Delete	remove()	delete

Searching an Array

```
// { _id: 1, courses: [ "CS472", "CS572", "CS435" ] }
// find all documents where courses value contains "CS572"
db.col.find({ courses: "CS572" })

// find all documents where courses value contains "CS472" or "CS572"
db.col.find({ courses: { $in: ["CS572", "CS472"] } })

// find all documents where courses value contains "CS472" and "CS572"
db.col.find({ courses: { $all: [ "CS572" , "CS472" ] } })
```

The **$in** operator selects the documents where the value of a field is an array that contains at any order any (at least one) of the specified elements

The **$all** operator selects the documents where the value of a field is an array that contains at any order all the specified elements

Search with $elemMatch

```
{ _id: 1, results: [ 1, 2, 5, 10 ] }
{ _id: 2, results: [ 5, 8, 9, 10 ] }
{ _id: 3, results: [ 10, 11, 12 ] }

db.test.find( { results: { $all: [ 5, 10 ] } } )
// scan results for value x times (accepts only value, no operators)
// All values must exist to return the document
// scan if it has 5 AND scan again if it has 10 = returns _id: 1


db.test.find( { results: { $elemMatch: { $gt: 5, $lt: 10 } } } )
// scan results array once: check if there is one value matches the condition
// _id: 2
```

$elemMatch

The **$elemMatch** operator matches documents that contain an array field with at least one **element** that matches **ALL** the specified query criteria.

```
{ _id: 1, results: [ 82, 85, 88 ] }
{ _id: 2, results: [ 75, 88, 89 ] }

db.scores.find( { results: { $elemMatch: { $gte: 80, $lt: 85 } } } ) // _id: 1

{ _id: 1, results: [ { product: "abc", score: 10 }, { product: "xyz", score: 5 } ] }
{ _id: 2, results: [ { product: "abc", score: 8 }, { product: "xyz", score: 7 } ] }
{ _id: 3, results: [ { product: "abc", score: 7 }, { product: "xyz", score: 8 } ] }

db.survey.find( { results: { $elemMatch: { product: "xyz", score: { $gte: 8 } } } } ) // _id: 3
```

Searching an Object

```
{_id: 1, email: { work: "work@mum.edu", personal: "personal@gmail.com"} }

// nothing will be returned
db.col.find({ email: { work: "work@mum.edu" } })
db.col.find({ email: { personal: "personal@gmail.com" } })
db.col.find({ email: { personal: "personal@gmail.com", work: "work@mum.edu"} })

// will work
db.col.find({ email: { work: "work@mum.edu", personal: "personal@gmail.com"} })

// how to search for one key only?
db.col.find({ "email.work": "work@mum.edu" })
db.col.find({ "email.personal": "personal@gmail.com" })
```

Update Methods

`db.collection.update()`

Either updates or replaces a single document that match a specified filter or updates all documents that match a specified filter.

`db.collection.updateOne()`

Updates at most a single document that match a specified filter even though multiple documents may match the specified filter.

`db.collection.updateMany()`

Update all documents that match a specified filter.

For more check the documentation

Field Update Operators

By default **`update()`** will: update a **single document** and **replace** everything but the `_id`

```
// Original document
{ "_id" : "1", "students" : 250, "courses" : ["CS572", "CS472"] }
db.col.update({_id:"1"}, {"students":500})
// Results
{ "_id" : "1", "students" : 500 }
```

```
// To target only one field use { $set: { field1: value1, ... } }
// If the field does not exist, $set will add a new field with the
   specified value
{ "_id" : "1", "students": 250, "courses" : ["CS572", "CS472"] }
db.col.update({_id:"1"}, {$set: {"students": 500, "entry": "Oct"} })
// Results
{ "_id" : "1", "students": 500, "courses" : ["CS572", "CS472"] , "entry":"Oct" }
```

Field Update Operators

```
// Original document
{ "_id" : "1", "students" : 250, "courses" : ["CS572", "CS472"] }
db.col.update({_id:"2"}, {"students":500}, {upsert: true})
// Results
{ "_id" : "1", "students" : 250, "courses" : ["CS572", "CS472"] }
{ "_id" : "2", "students" : 500 }
```

```
// update all docs to have one more field (city: Fairfield)
{ "_id" : "1", "Dept" : "CompPro" }
{ "_id" : "2", "Dept" : "SustainableLiving" }
db.col.update({}, {$set: {"city":"Fairfield"}} , {multi: true})
// Results
{ "_id" : "1", "Dept" : "CompPro", "city":"Fairfield" }
{ "_id" : "2", "Dept" : "SustainableLiving", "city":"Fairfield" }
```

Field Update Operators

Update Operator	Description	Notes
`$set`	Replaces the value of a field with the specified value	If the field does not exist, `$set` will add a new field with the specified value
`$inc`	Increments a field by a specified value, it accepts positive and negative values	If the field does not exist, `$inc` creates the field and sets the field to the specified value
`$unset`	Deletes a particular field, The specified value in the $unset expression does not impact the operation.	If the field does not exist, then `$unset` does nothing

Examples - Field Update Operators

```
{ "_id" : "1", "students" : 250 }
db.col.update({_id:"1"}, { $inc : { "students":1, "exams":1 } })
{ "_id" : "1", "students" : 251, "exams":1 }
```

```
{ "_id" : "1", "students" : 250, "dept": "ComPro" }
db.col.update({_id:"1"}, { $unset : { "dept":1 } })
{ "_id" : "1", "students" : 250 }
```

Array Update Operators

```
//Original Document { "_id" : 1, "a" : [1, 2, 3, 4] }
db.testCol.update({_id:1}, { $set : { "a.2":5 } })
// output: { "_id" : 1, "a" : [1, 2, 5, 4] }
db.col.update({_id:1}, { $push : { "a": 6 } })
// output: { "_id" : 1, "a" : [1, 2, 5, 4, 6] }
db.col.update({_id:1}, { $pop : { "a": 1 } })
// output: { "_id" : 1, "a" : [1, 2, 5, 4] }
db.col.update({_id:1}, { $pop : { "a": -1 } })
// output: { "_id" : 1, "a" : [2, 5, 4] }
db.col.update({_id:1}, { $pushAll : { "a": [7, 8, 9] } })
// output: { "_id" : 1, "a" : [2, 5, 4, 7, 8, 9] }
db.col.update({_id:1}, { $pull : { "a": 5 } })
// output: { "_id" : 1, "a" : [2, 4, 7, 8, 9] }
db.col.update({_id:1}, { $pullAll : { "a": [2, 4, 8] } })
// output: { "_id" : 1, "a" : [7, 9] }
db.col.update({_id:1}, { $addToSet : { "a": 5 } })
// output: { "_id" : 1, "a" : [7, 9, 5] }
db.col.update({_id:1}, { $addToSet : { "a": 5 } })
// output: { "_id" : 1, "a" : [7, 9, 5] }
```

$ (The Array POSITION that matched)

The positional **$** operator identifies an **element** in an array to **update** without explicitly specifying the position of the element in the array.

```
{ "_id" : 1, "grades" : [ 85, 80, 84 ] }
db.students.updateOne( { _id: 1, grades: 80 }, { $set: { "grades.$" : 82 } } )
// { "_id" : 1, "grades" : [ 85, 82, 84 ] }

{ _id: 4, grades: [ { total: 80, mean: 75, std: 8 },
                    { total: 85, mean: 90, std: 5 },
                    { total: 86, mean: 85, std: 8 } ] }
db.students.updateOne( { _id: 4, "grades.total": 85 },
                       { $set: { "grades.$.std" : 6 } } )
// { "total" : 85, "mean" : 90, "std" : 6 }
```

You must include the array field as part of the query.

$elemMatch vs. arrayFilters

```
{ _id: 5, grades: [ { total: 80, mean: 75, std: 8 },
                    { total: 85, mean: 90, std: 5 },
                    { total: 90, mean: 85, std: 3 } ] }

db.students.updateOne(
  { _id: 5, grades: { $elemMatch: { total: { $lte: 90 }, mean: { $gt: 80 } } } },
  { $set: { "grades.$.std" : 6 } }
)
```

```
{ total: 85, mean: 90, std: 6}
Only updates one element that matches our condition
```

```
db.students.updateOne(
   { _id: 5 },
   { $set: { "grades.$[obj].std" : 6 } } ,
   { arrayFilters: [{ "obj.total": { $lte: 90 } ,"obj.mean": { $gt: 80 } }] }
)
```

```
{ total: 85, mean: 90, std: 6 }, { total: 90, mean: 85, std: 6 }
Always updates all elements in the array that match our condition
```

15

Using arrayFilters

```
{ "_id" : 1,
  "grades" : [
        { type: "quiz", questions: [ 10, 8, 5 ] },
        { type: "quiz", questions: [ 8, 9, 6 ] },
        { type: "hw", questions: [ 5, 4, 3 ] },
        { type: "exam", questions: [ 25, 10, 23, 0 ] },
] }
```

```
db.students.update(
  {},
  { $inc: { "grades.$[t].questions.$[score]": 2 } },
  { arrayFilters: [ { "t.type": "quiz" } , { "score": { $gte: 8 } } ],
    multi: true }
)
```

```
{ "type" : "quiz", "questions" : [ 12, 10, 5 ] },
{ "type" : "quiz", "questions" : [ 10, 11, 6 ] },
```

16

elemMatch vs arrayFilters

```
{grades:[10,20,20]}
{grades:[10,20,20]}

db.test.update({grades: { $elemMatch: { $gt: 10 , $lt: 30 } }},
               {$set:{'grades.$': 15}},
               {multi: true})
{ "grades" : [ 10, 15, 20 ] }
{ "grades" : [ 10, 15, 20 ] }

db.test.update({},
               {$set:{'grades.$[c]': 30}},
               { arrayFilters: [ { "c": {$gt: 10 , $lt: 30} } ], multi: true})

{ "grades" : [ 10, 30, 30 ] }
{ "grades" : [ 10, 30, 30 ] }
```

Multi-updates

What really happen when we do multi-documents write operations is that inside MongoDB there is a **single thread** for every operation being executed (run sequentially inside single thread). Every write operation that affects more than one document is carefully coded in multi-tasking way to occasionally **yield** control to allow other operations to work on the same dataset.

- However, MongoDB guarantee an **individual** doc to be **atomic**, so no other R/W operation can get a half updated document.
- Multi-doc update operations are **not isolated transactions**. Check `$isolated()`

Performance

When we talk about performance we think adding memory, replacing the hard drive with SSD or using more powerful CPU (vertical scaling).

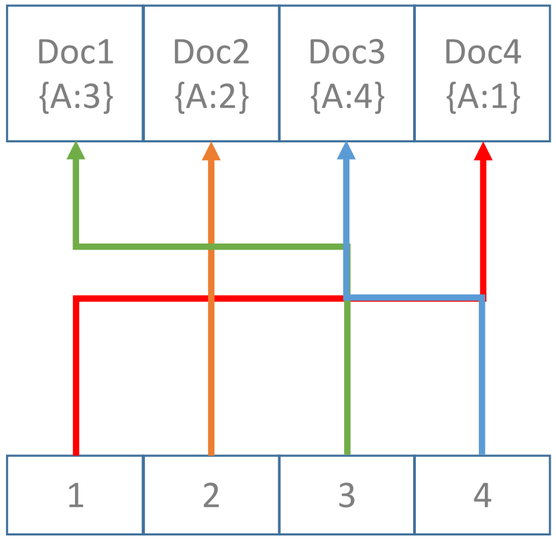
But what really affects performance is our algorithm and the logic we use in structuring our DB!

Understanding indexes leads to a better code and performance.

When would MongoDB use an Index?

Regular Index

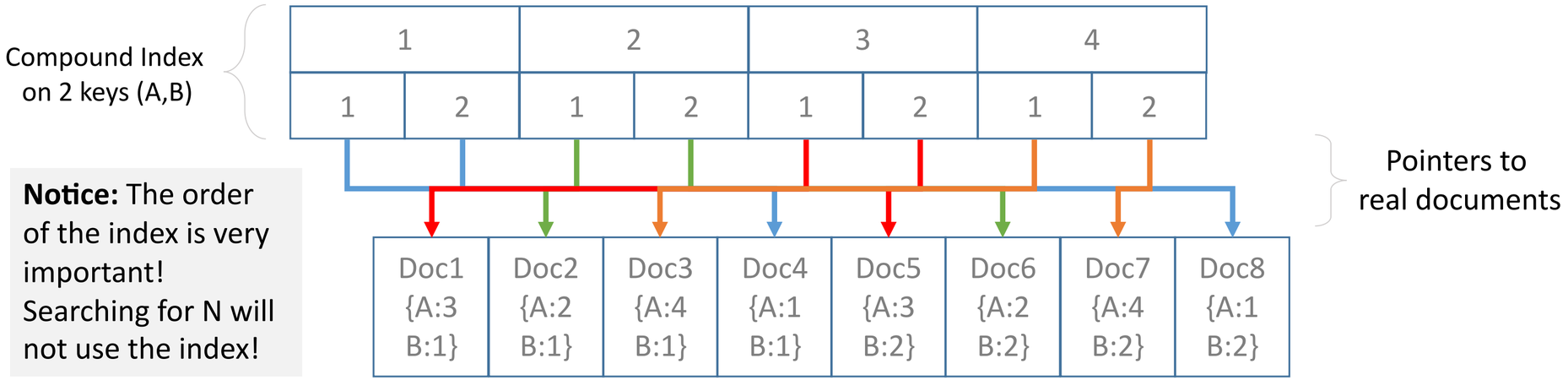
- When we call **`find()`** a **full table/collection scan** will happen because our collection is not sorted which leads to slow performance.
- That's why we create an Index to boost the performance of our operations *(find, update, sort.. etc)*



Index on A key

Compound Index

When the index is composed of more than one key.



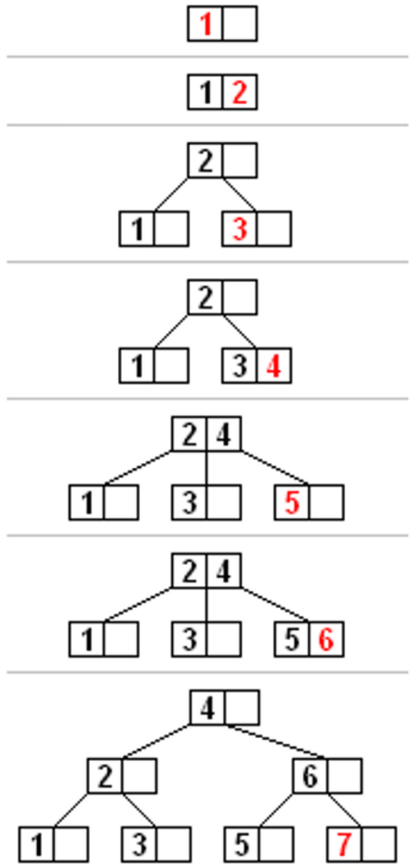
Notice: The order of the index is very important! Searching for N will not use the index!

Notice: The index will be updated for every `insert()` operation (slow). Though reading from the index is very fast.

B-Tree Algorithm

MongoDB uses B-Tree algorithm to maintain indexes. B-tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.

Algorithm	Average	Worst Case
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)



Index Operations

```
// create a compound index on Key1 ASC, Key2 DESC
db.colName.createIndex({key1:1, key2:-1})

// show a list of all indexes in the collection
db.colName.getIndexes()

// drop the index
db.colName.dropIndex({key1:1, key2:-1})

// show useful information about index size
db.colName.stats()
```

Note: Creating an index will take time and space!

MultiKey Index

MongoDB supports index for **Array typed keys**. Only one Array typed key is allowed and MongoDB will create a compound key for all value combinations.

```
{
  name: '',
  sports: ['cycling', 'swimming'],
  address: [ {type: 'home', location: ['IA', 'Fairfield']},
             {type: 'work', location: ['NY', 'New York']} ]
}
db.colName.createIndex({name:1, sports:1})
db.colName.createIndex({name:1, address:1})
db.colName.createIndex({'address.type':1}) // supports rich documents
db.colName.createIndex({'address.location':1})
db.colName.createIndex({sports:1, address:1}) // will fail
```

Examples – MultiKey Index & Covered Index

```
db.col.insert({name:'Asaad' , email: 'asaad@mum.edu' , work:'MUM'})
db.col.createIndex({name:1, email:1}) // Create Compound Index
db.col.find({name:'Asaad'}).explain() // Not Covered
db.col.find({name:'Asaad'}, {name:1, email:1, _id:0}).explain() // Covered Index

db.col.insert({name:['Mike','Mada'], email:'email@mum.edu'})
db.col.find({name:'Mike'}).explain() // MultiKey Index

// Will fail: cannot index parallel arrays
db.col.insert({name:['George','Angel'], email:['george@mum.edu','angel@mum.edu']})
```

Covered Index: The index is enough to return the results, there is no need to go to the original document

Unique Index

```
db.col.insert({a:1, b:1})
db.col.insert({a:2, b:2})

// create a unique index on "a"
db.col.createIndex({a:1},{unique: true})

db.col.insert({a:2, b:2}) // will fail
```

Sparse Index

```
db.col.insert({email:'asaad@mum.edu'})
db.col.insert({email:'mada@mum.edu'})
db.col.insert({name:'Mike', email:'mike@mum.edu'})
// create an index on "name"
db.col.createIndex({name:1}) // will work
// create a unique index on "name"
db.col.createIndex({name:1},{unique: true}) // will fail duplicate "null"
// create a unique index that doesn't have reference to the first two documents!
db.col.createIndex({name:1},{unique: true, sparse: true})

db.col.find({name:{$exists: false}}) // will use BasicCursor and return 2 docs
db.col.find({name:{$exists: false}}).hint({name:1}) // will use "name" index and
return nothing
```

Sparse indexes only contain entries for documents that have the indexed field. The index skips over any document that is missing the indexed field. The index is “sparse” because it does not include all documents of a collection.

Create Index in Production!

Creating an index for a large collection is a process that will block all other IO to the collection. MongoDB creates indexes in the foreground by default, despite it's a faster option, blocking other IO might not be an option for systems in production.

Creating an index in the background (writes can happen concurrently) is more efficient, but it's way more slower than foreground process.

```
db.collection.createIndex( { a: 1 }, { background: true } )
```

Is there another way to create an index fast in production?

How MongoDB chooses which index to use?

Let’s say we have 3 indexes for our collection, the first time we perform a query, MongoDB will run three separate instances of the query using all three indexes, the first one finishes will be saved as the fastest index for this kind of queries. When data changes, MongoDB will run such a test again.

Remember we can always hint MongoDB to use a certain index:

```
// in Mongo Shell
db.test.find({name:'Asaad'}).hint({name:1})
```

```
// in Node.js
db.collection('test').find({name:'Asaad'}, {'hint':{'name':1}})
```

Index Cardinality

For better performance, **it's important to keep indexes in memory**. Always check your DB stats **`db.colName.stats()`** and make sure you have enough memory for indexes.

In proportion to collection size, index cardinality will be:

- **Regular Index** 1:1
- **Sparse Index** less or equal to documents size
- **MultiKey Index** significantly more than documents size

When you realize that you don't have enough resources for your index, it's time to consider Sharding.

Index Selectivity

It’s important to understand our data before we select an index. Let’s say we want to keep a log of activities in our application, **choosing an index with more varieties** will cut the time of searching and make our operations more efficient.

type	date
save	12 sep 2019
update	12 sep 2019
update	13 sep 2019
save	13 sep 2019
save	13 sep 2019

An index of (type, date) is not a good option as type is limited. While an index of (date, type) will improve the system performance because we can reach to the documents much faster

FullText Index

When looking for a string value in MongoDB the full string will be checked against. While regular expressions could be a solution, it’s very slow because it will search the entire collection. FullText Index is the best way to build an index for all words.

```
db.col.insert({mytext:'Hello goes to CS572 Class!'})
db.col.find({mytext:'Hello CS572 Class'}) // will find 0 docs
db.col.find({mytext:'CS572'}) // will find 0 docs
db.col.find({mytext: {$regex: 'CS572'} }) // will find 1 doc but slow!

db.col.createIndex({mytext:'text'})
// will find our document fast using FullText index. spaces treated as OR
db.col.find({$text: {$search:'Class CS572'} })
```

Note: MongoDB will log out all slow queries (more then 100ms) automatically to the shell. To write the log on the disk use a Profiler

A More Relevant FullText Search Results

FullText search will calculate a **textScore** value automatically for me, we can add this field to our results and sort them based on that field to get the documents that are more relevant.

```
{$text: {$search:'text1 text2'} }
```

```
db.collection('colName').find({query}, { score: {$meta:'textScore'}} )
                        .sort( { score: {$meta:'textScore'}} )
```

The { $meta: "textScore" } expression provides information on the processing of the $text operation.

Distance Between 2 Points - The Haversine Formula

For two points on a sphere (of radius R) with latitudes ϕ1 and ϕ2, latitude separation Δϕ = ϕ1 − ϕ2, and longitude separation Δλ the distance d between the two points

$$\operatorname{haversin}\left(\frac{d}{R}\right) = \operatorname{haversin}(\Delta\phi) + \cos(\phi_1)\cos(\phi_2)\operatorname{haversin}(\Delta\lambda)$$

$$\operatorname{haversin}(\theta) = \frac{\operatorname{versin}(\theta)}{2} = \sin^2\left(\frac{\theta}{2}\right)$$

$$\operatorname{versin}(\theta) = 1 - \cos(\theta) = 2\sin^2\left(\frac{\theta}{2}\right)$$

The Haversine Formula in MySQL

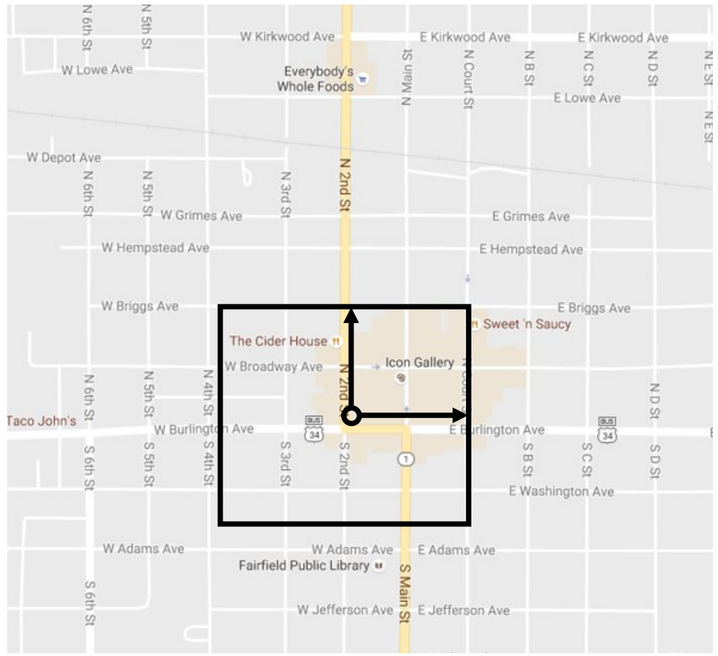
SELECT *, *3956 * 2 * ASIN(SQRT(POWER(SIN((@orig_lat - abs(dest.lat)) * pi()/180 / 2), 2) + COS(@orig_lat * pi()/180) * COS(abs(dest.lat) * pi()/180) * POWER(SIN((@orig_lon – dest.lon) * pi()/180 / 2), 2)))* as distance **FROM** hotels dest **HAVING** distance < @dist **ORDER BY** distance **LIMIT** 10;

PREV NEXT

Geospatial Indexes (2D)

To find nearest locations in MongoDB all we need is :

- Have all locations saved in each collection with [long, lat] array
- Create an Geospatial Index (2d) on that key.

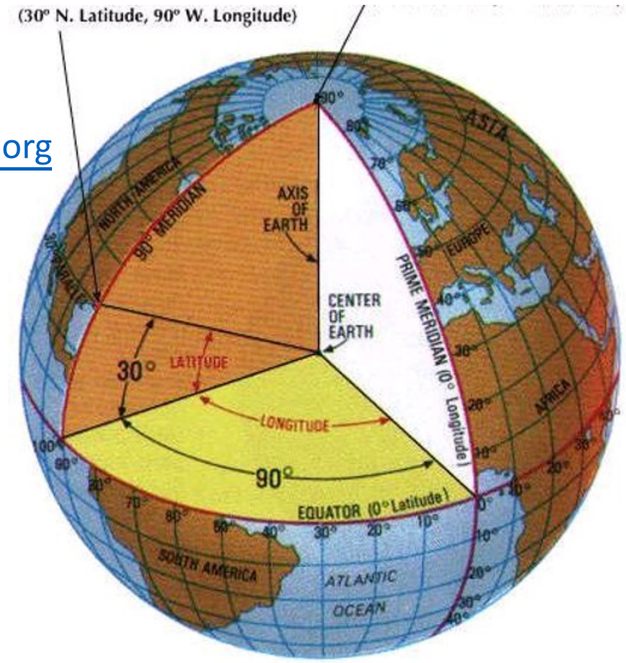


Code Example:

```
db.collection.insert{name:'Restaurant', location: [Long,Lat]}
db.collection.createIndex({location: '2d'})
// results will be sorted by increasing distance (closest to furthest)
db.collection.find({location: {$near:[currentLong, currentLat]}}).limit(10)
```

Geospatial Spherical (3D)

Searching on spherical surface is more accurate and gives better results.
You can check encoding of geographic data structures at http://geojson.org




```
{ "_id": 1,
  "name": "MUM University",
  "city": "Fairfield",
  "location": {
        "type":"Point",
        "coordinates": [-91.9612747,41.0132949] }
}

db.places.createIndex({'location':'2dsphere'})
db.places.find({location:{
       $near:{$geometry:{type: "Point", coordinates:[-91.9627755,41.0085809]},
              $maxDistance:2000 } } })
```

In meters