Release: 20250221

Multikey Indexes

- A multikey index is an index that has an array as one of the fields being indexed
  - Can index primitives, documents, or sub-arrays
  - Are created using `createIndex()` just as non-multikey indexes
- An index entry is created on each unique value found in an array

Multikey indexes are indexes on an array.

They are created using the same syntax as normal indexes

You cannot create a compound multikey index if more than one to-be-indexed field of a document is an array.

Exercise - Multikey Basics

Exercise for the class:

How many documents are inserted?

How many index entries are there in total for **lap_times**?

How many documents do the queries find?

Do they use the index?

```
MongoDB> use test

MongoDB> db.race_results.drop()

MongoDB> db.race_results.createIndex({"lap_times": 1})

MongoDB> db.race_results.insertMany([
      { "lap_times" : [ 3, 5, 2, 8 ] },
      { "lap_times" : [ 1, 6, 4, 2 ] },
      { "lap_times" : [ 6, 3, 3, 8 ] }
  ])

// Answer Questions before running the following:

MongoDB> db.race_results.find( { lap_times : 1 } )

MongoDB> db.race_results.find( { "lap_times.2" : 3 } )
```

Array of Documents

How many documents are inserted in total?

For each query:

How many results we get?

Which index, if any, is being used?

```
MongoDB> db.blog.drop()

MongoDB> db.blog.insertMany([
      {"comments": [{ "name" : "Bob", "rating" : 1 },
                    { "name" : "Frank", "rating" : 5.3 },
                    { "name" : "Susan", "rating" : 3 } ]},

      {"comments": [{ name : "Megan", "rating" : 1 } ] },

      {"comments": [{ "name" : "Luke", "rating" : 1.4 },
                    { "name" : "Matt", "rating" : 5 },
                    { "name" : "Sue", "rating" : 7 } ] }
])

MongoDB> db.blog.createIndex( { "comments" : 1 } )
MongoDB> db.blog.createIndex( { "comments.rating" : 1 } )

// Answer Questions before running the below queries

MongoDB> db.blog.find({ "comments":{"name":"Bob","rating":1}})

MongoDB> db.blog.find({ "comments": {"rating" :1} } )
MongoDB> db.blog.find({ "comments.rating":1} )
```

Arrays of Arrays

How many documents are inserted in total?

For each query:

How many results we get?

Which index, if any, is being used?

```
MongoDB> db.player.drop()
MongoDB> db.player.createIndex( { "last_moves" : 1 } )
MongoDB> db.player.insertMany([
 { "last_moves" : [ [ 1, 2 ], [ 2, 3 ], [ 3, 4] ] },
 { "last_moves" : [ [ 3, 4 ], [ 4, 5 ] ] },
 { "last_moves" : [ [ 4, 5 ], [ 5, 6 ] ] },
 { "last_moves" : [ [ 3, 4 ] ] },
 { "last_moves" : [ [ 4, 5 ] ] }
])

// Answer Questions before running below queries

MongoDB> db.player.find( { "last_moves" : [ 3, 4 ] } )
MongoDB> db.player.find( { "last_moves" : 3 } )
MongoDB> db.player.find( { "last_moves.1" : [ 4, 5 ] } )
```


Note: This type of index on arrays can quickly consume memory

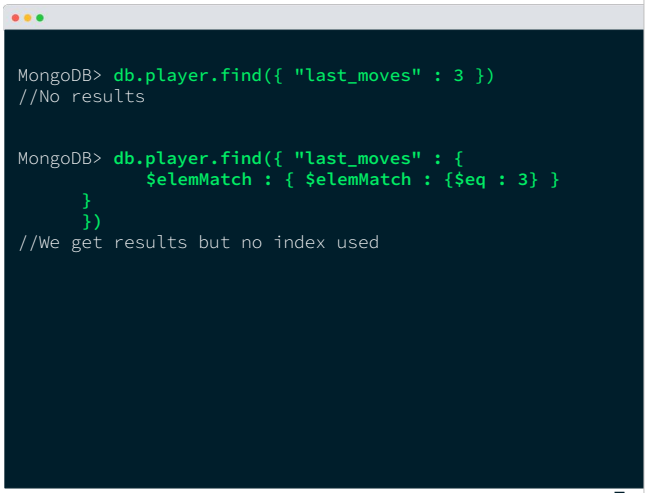
Nested Queries

Recall $elemMatch

Find where an array member matches the query

Elements in a nested array cannot be indexed

How can we change the structure to use an index?



On the Previous page we tried to search for a value in an array of arrays - and not only could we not index it but there seemed to be no way to search for it.
If you do need to search in an array of arrays it is possible - but using the very powerful if misunderstood $elemMatch which returns true or false based on an array member matching a query.

Ans: We cannot use an index to support an dynamic (random) embedded array. The structure would have to be a 1D array in order to have support from a multikey index. Alternatively, if the 2d array is not dynamic, or we are only looking inside of a specific position in the 2d array such as [0, 0], then we can use a partialfilterexpression during the index creation such as the following: db.player.createIndex({"last_moves.0.0": 1}, {partialFilterExpression: {"last_moves.0.0": {$eq: 3}}}) - this creates a single field index in the second scenario.

Compound Indexes

Compound indexes are indexes based on more than one field

- The most common type of index
- Same concept as indexes used in an RDBMS
- Up to 32 fields in a compound index
- Created like single-field index but specifying all the index fields

The field order and direction is very important

```
db.people.createIndex({lastname:1, firstname:1, score:1})
```


Compound Indexes can support queries that match multiple fields
MongoDB Will not use two indexes together in a query except to support $or clauses where all branches of the $or have indexes
They should be used instead of creating multiple single indexes
Limit of 32 fields per index

Compound Indexes (Cont.)

Can be used as long as the first field in index is in the query

Other fields in the index do not need to be in the query

Example:

- `...createIndex({country:1, state:1, city:1})`
- `...find({country:"UK", city:"Glasgow"})`

The order of fields in a compound index is important.

In addition to supporting queries that match all the index fields, compound indexes can support queries that match on the prefix of the index fields.

Compound Indexes (Cont.)

MongoDB uses the index for country and city but must look at every state in the country, so looks at many index keys.

A Better Index for this query would be this:

- `...createIndex({country:1, city:1, state:1})`

Field Order Matters

Equality First

- What fields, for a typical query, are filtered the most
- Selectivity is NOT cardinality, selective can be a boolean choice
- Normally Male/Female is not selective (for the common query case)
- Dispatched versus Delivered IS selective though

Then **S**ort and **R**ange

- Sorts are much more expensive than range queries when no index is used
- The directions matter when doing range queries

What use cases would the following Compound Index support?

```
db.orders.createIndex({order_status:1, postcode:1, order_date:-1, number_of_items:
-1})
```


In compound indexes, the goal is to reduce the size of the search tree as early as possible by being as selective as possible.

Consider your application's use cases and queries. Let's say your delivery app needs to show users where their order currently is. This means you are mostly interested in orders which are currently 'dispatched'. You don't want to search through all the orders which have already been 'delivered'.

Sort field examples might be 'score' or 'last_name'

Range field examples might be 'date_created' or 'distance_travelled'. If you are more interested in records where 'date_created' is recent, descending order would be most helpful.

You may need to experiment to find the optimal combination and order of fields for your most important use cases. Remember to limit the number of indexes per collection.

Use Case:
Message Board App

A Simple Message Board

Look at the indexes needed for a simple Message Board App:

The app automatically cleans up the board removing older, low-rated anonymous messages on a regular basis.

Query requirements:

1. Find all messages in a specified timestamp range
2. Select for whether the messages are anonymous or not
3. Sort by rating from lowest to highest

Message Board Index

Create five messages (just with the relevant fields):

timestamp

username

rating

Index on the **timestamp**

```
MongoDB> msgs = [
{ "timestamp": 1, "username": "anonymous", "rating": 3},
{ "timestamp": 2, "username": "anonymous", "rating": 5},
{ "timestamp": 3, "username": "sam", "rating": 2 },
{ "timestamp": 4, "username": "anonymous", "rating": 2},
{ "timestamp": 5, "username": "martha", "rating" : 5 }
]

MongoDB> db.messages.drop()

MongoDB> db.messages.insertMany(msgs)


//Index on timestamp
MongoDB> db.messages.createIndex({ timestamp : 1 })
```

Message Board Index

explain() shows good performance when filtering by **timestamp**, but this is not the whole query

Need to filter for **anonymous** users too

```
MongoDB> db.messages.find({
            timestamp : { $gte : 2, $lte : 4 }
      }).explain("executionStats")

...

      executionStats: {
            executionSuccess: true,
            nReturned: 3,
            executionTimeMillis: 0,
            totalKeysExamined: 3,
            totalDocsExamined: 3,

...
```


Example - A Simple Message Board

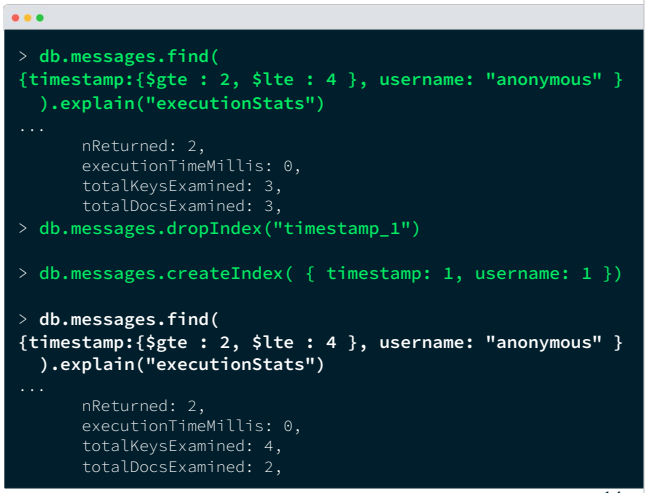
Message Board Index

Not as efficient as it could be:

totalKeysExamined > nReturned

What if username is added to the index?

Is there an improvement?



```
> db.messages.find(
{timestamp:{$gte : 2, $lte : 4 }, username: "anonymous" }
  ).explain("executionStats")
...
      nReturned: 2,
      executionTimeMillis: 0,
      totalKeysExamined: 3,
      totalDocsExamined: 3,
> db.messages.dropIndex("timestamp_1")

> db.messages.createIndex( { timestamp: 1, username: 1 })

> db.messages.find(
{timestamp:{$gte : 2, $lte : 4 }, username: "anonymous" }
  ).explain("executionStats")
...
      nReturned: 2,
      executionTimeMillis: 0,
      totalKeysExamined: 4,
      totalDocsExamined: 2,
```


Example - A Simple Message Board

We are dropping the previously created index “timestamp_1” because if we don’t do so, the priority would still be given to that index despite of having the newly created index in place.

Index fields in wrong order

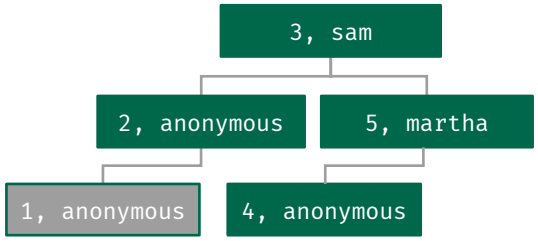
```
Query: {timestamp:{$gte:2, $lte:4}, username:"anonymous"}

Index: { timestamp: 1, username: 1 }
```

Range first as the `timestamp` is first in the index

- Start at the first `timestamp` and check if >=2
- Walk tree Left to Right until not <= 4 (3 nodes)
- Check each of those 3 nodes:=`'anonymous'`
- Return only 2 nodes

Summary: 2 returned and 4 visited



Index order matters as demonstrated in the example.

Index fields in correct order

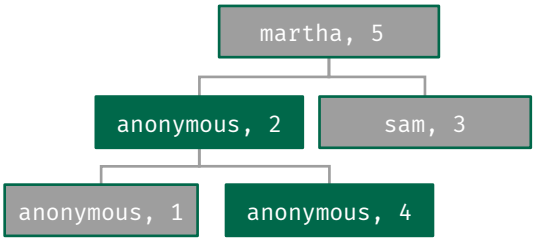
```
Query: {timestamp:{$gte:2, $lte:4}, username:"anonymous"}

Index: { username: 1, timestamp: 1 }
```

Equality first as the **username** is first in the index

- Exact match filters the tree to walk (3 **anonymous**)
- Find first **anonymous** where **timestamp** >=2
- Walk tree while **anonymous** & **timestamp** <=4
- Return 2 nodes

Summary: 2 returned and 2 visited

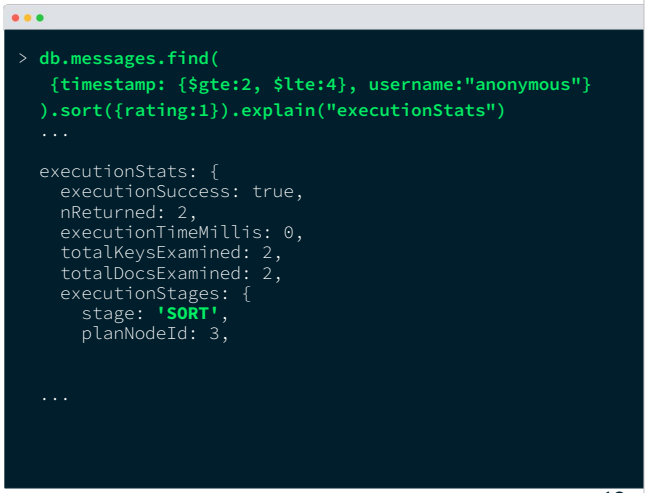


Indexing in the correct order will achieve improved results.

Indexing for Sort

Sort by rating from lowest to highest

Sort direction usually not important as we can walk the index tree forwards or backwards



The index should also cover sorting where possible to prevent sorting in memory.

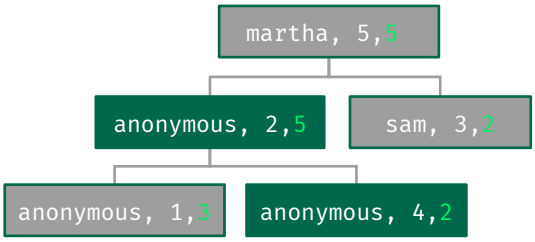
Index in correct order?

```
Query: {timestamp:{$gte:2, $lte:4}, username:"anonymous"}
Sort: { rating: 1 }
Index: { username: 1, timestamp: 1 , rating: 1}
```

Equality first as the **username** is first in the index

- Exact match filters the tree to walk (3 **anonymous**)
- Find first **anonymous** where **timestamp** >=2
- Walk tree while **anonymous** & **timestamp** <=4
- Return 2 nodes but order is 5,2
- Need to sort results

Summary: 2 returned and 2 visited + sort stage



Adding a sort field to the index after a range can produce undesired results.

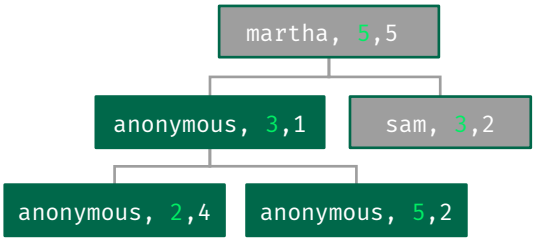
Index in better order?

```
Query: {timestamp:{$gte:2, $lte:4}, username:"anonymous"}
Sort: { rating: 1 }
Index: { username: 1, rating: 1, timestamp: 1 }
```

Equality first as the `username` is first in the index

- Exact match filters the tree to walk (3 `anonymous`)
- Find first `anonymous`
- Walk tree while `anonymous`
- Check where `timestamp` >=2 & `timestamp` <=4
- Return 2 nodes and rating is ordered: 2, 5

Summary: 2 returned and 3 visited



Avoiding a SORT stage is preferable even if it increases the number of keys which must be examined.

Rules of Compound Indexing

The generic rule for ordering fields in a compound index:

1. **E**quality
2. **S**orting
3. **R**ange

The general rule is Equality, Sort, Range.
Note this is only a generic guideline, there are cases where Range may be better placed before Sort.

MultiKey Compound Indexes

An index can be Compound (multiple fields) and MultiKey (multiple values for a field)

In a Document any non _id field can be an array

Indexing two arrays in one document returns an error as MongoDB would need to store all possible combinations

```
> use test
switched to db test

> db.multikey.createIndex({ a:1, b:1, c:1})
a_1_b_1_c_1

> db.multikey.insertOne({a:"temps",b:[1,2,3],c:4})
{
  acknowledged: true,
  insertedId: ObjectId("...")
}


> db.multikey.insertOne({a:"temps",b:2,c:[8,9,10]})
{
  acknowledged: true,
  insertedId: ObjectId("...")
}

> db.multikey.insertOne({a:"temps",b:[2,3],c:[8,9,10]})
MongoServerError: cannot index parallel arrays [c] [b] ✗
```


You can have a compound Multikey index, all the fields in the compound index are stored in an index entry for each unique value in the array field.
For any given document - which field is an array in the compound index does not matter.
But only one of the fields in any give compound index can be an array in a single document.
If this was not the case we would need an index entry for every possible combination of values - which might be huge.

Index covered queries

Fetch all data from Index, not from the Document

Are much faster in some cases

Limitations to be aware of:

- Use projection { _id : 0 } if _id is not part of the index
- Cannot cover queries when querying over arrays or arrays are returned

```
> use test
switched to db test

> db.messages.find(
   {timestamp: {$gte:2, $lte:4}, username:"anonymous"},

   {_id: 0, timestamp: 1, username: 1, rating: 1}
  ).sort({rating:1}).explain("executionStats")
...
  executionStats: {
    executionSuccess: true,
    nReturned: 2,
    executionTimeMillis: 0,
    totalKeysExamined: 2,
    totalDocsExamined: 0,
    executionStages: {
      stage: 'PROJECTION_COVERED',
      planNodeId: 3,
```


If all the fields we need can be found in the index then we don't need to actually fetch the document.

We need to remove _id from the projection if it's not in the index we use to query.

We cannot use a multikey index for projection as we don't know from the index entry about position or quantity of values, or even if it's an array versus a scalar value in any given document.

Indexes store data in a slightly different format to BSON as all numbers have a common format in the index (and a type) and so need to be converted back to BSON , this takes more processing time.

If we are fetching one or two values form a larger document - index covering is good - if it's most of the fields in a document it's probably better to fetch from the document.

If we need to add extra fields to the index in order to facilitate covering, add them at the end and be aware of the extra storage.

Index-covered queries are the ultimate goal but not always achievable.

Additional Example: https://www.mongodb.com/docs/v7.0/core/query-optimization/#std-label-covered-queries

Exercise - Compound indexes

Create the best index you can for this query - how efficient can you get it?

```
> query = { amenities: "Waterfront",
  "bed_type" : { $in : [ "Futon", "Real Bed" ] },
  first_review : { $lt: ISODate("2018-12-31") },
  last_review : { $gt : ISODate("2019-02-28") }
  }
> project = { bedrooms:1 , price: 1, _id:0, "address.country":1}
> order = {bedrooms:-1,price:1}
> use sample_airbnb
> db.listingsAndReviews.find(query,project).sort(order)
```

Time to Live (TTL) Indexes

Not a special Index, just a flag on an index on a date

MongoDB automatically deletes documents using the index based on time

Background thread in server runs regularly to delete expired documents

```
> db.t.drop()

> db.t.insertOne({ create_date : new Date(),
user: "bobbyg",session_key: "a95364727282",
cart : [{ sku: "borksocks", quant: 2}]})

> db.t.find()

> db.t.createIndex({"create_date":1}, {expireAfterSeconds
: 60 })
> //TTL set to auto delete where create_date >1 minute

> for(x=0;x<10;x++) { print(db.t.countDocuments()) ; slee
p(10000) }
```


TTL indexes allow data to be deleted when it expires
The expiration period is set when creating the index

Time to Live (TTL) Indexes

Alternative way to use TTL Indexes

- Put the date you want it deleted in a field (expireAt)
- Add a TTL with expireAfterSeconds set to 0

Watch out of unplanned write load

- Better to write your own programmatic data cleaner.
- Schedule using a framework like cron/scheduler/Atlas

TTL indexes should be used with caution as restoring deleted data can be a huge pain or even not possible if no backups exist.
These should be widely communicated to ensure it comes as no surprise when this happens.
Be careful not to delete huge amounts of data in production.

Index usage tips

Indexes add overhead to write operations per index entry (beware of multikey!)

Accessing Index should be from RAM based cache, if it is not in cache, MongoDB fetches data from disk

Can use $indexStats() aggregation to see how much each index is used

More Indexes generally means more RAM required

- Remove indexes that aren't used by any queries to save CPU and RAM.
- If one index is a prefix of another it is redundant `{ a: 1}` and `{a:1,b:1}`

Indexes are mostly just like RDBMS indexes - but queries are simpler

Indexing should be used effectively to improve query performance, but over-indexing or indexing incorrectly can have an adverse effect as write operations might have some overhead updating the index values - this depends on the application but it can be anywhere from 5% to 70%.
Indexes do NOT need to be held entirely in RAM - like collections the database will cache in RAM parts it accesses often and evict those it doesn't.
The index exists on DISK with a cache in RAM of parts accesses recently - we want to access something from RAM so we need to design the indexes
so infrequently accessed data does not get cached (or add more RAM). Often this is based on a date value in the index.

Quiz Time!

#1. What things might prevent an index to 'cover' a query?

A	Projecting the _id field	B	Having a multikey index	C	Sorting query results by one of the index fields
D	Retrieving an array field	E	Having more than one index on the same field		

Answer in the next slide.

#1. What things might prevent an index being used to 'cover' a query?

A. Projecting the _id field

B. Having a multikey index

C. Sorting query results by one of the index fields

D. Retrieving an array field

E. Having more than one index on the same field

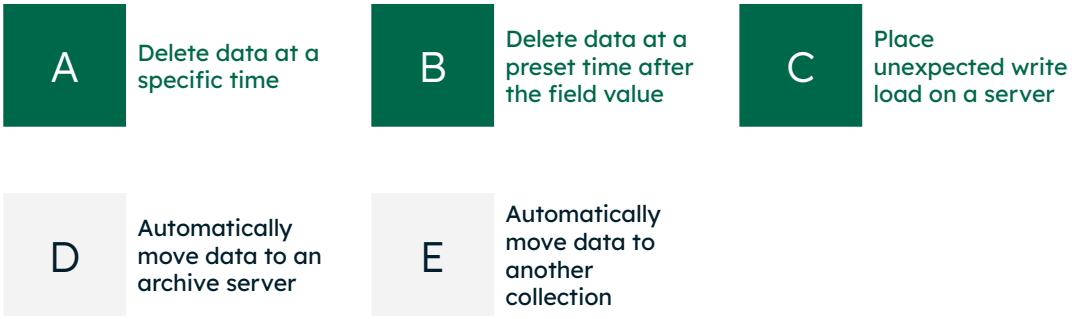
Indexing on array fields creates a multikey index which doesn't have the functionality to cover a query like a compound index would. Retrieving an array field from a compound index is also a limitation that will not yield a COVERED_PROJECTION stage and the _id is alway returned by default so best to also project it out. A,B,D will prevent an index covered query.

#2. Select three actions TTL Indexes can perform:

A	Delete data at a specific time
B	Delete data at a preset time after the field value
C	Place unexpected write load on a server
D	Automatically move data to an archive server
E	Automatically move data to another collection

Answer in the next slide.

#2. Select three actions TTL Indexes can perform:



Time To Live (TTL) are created to delete data, not move or archive. The configuration can be done at a specific time or a preset time after the timestamp field value - in either scenario, it will cause a write load on the db when the background thread executes to delete said documents.

Recap

Compound indexes are the most used ones in MongoDB

The order in a compound index is very important

Index covered queries greatly reduce I/O cost

Appendix

Geospatial Indexing

Most indexed fields are Ordinal (A < B < C) and can be sorted - Sorted data can be efficiently searched with a binary chop O(log n)

Locations - coordinates:

- Can be sorted East <-> West OR North<-> South - But not both
- They require a totally different indexing concept (Geohashes or ZTrees)
- MongoDB uses Geohashes and has a very comprehensive geospatial capability

Geospatial Indexing requires a totally different indexing concept.
Geohash transforms 2d coordinate values into 1d key space values, which are then stored in normal MongoDB B-trees.

Geospatial types

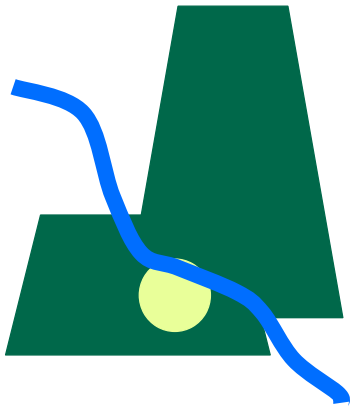
Geospatial data has its own types

- Points
- Lines
- Circles
- Polygons

Groups of the above additive and subtractive

Written as **GeoJSON** or `[longitude,latitude]`

```
area = {type : "Polygon",
        coordinates : [
      [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ],
      [ [ 2 , 2 ] , [ 3 , 3 ] , [ 4 , 2 ] , [ 2 , 2 ] ]
   ]}
```



Geospatial data has its own types

Geospatial Capabilities

Geo Indexes quickly determine geometric relationships:

- All shapes within a certain distance of another shape
- Whether or not shapes fall within another shape
- Whether or not two shapes intersect

Many Use cases:

- Find all bars near here
- Find all motels along my route
- Find all counties that are impacted by this hurricane

Queries can be combined with other predicates - Find all motels on my route with a pool

Geo Indexes can be compound with other fields

Geospatial has many use cases in modern applications.

Coordinate spaces

MongoDB can use **Cartesian (2d)** or **Spherical Geometry (2dsphere)**

Cartesian:

- Simple Lat/Long -90 to 90 and -180 to 180 degrees
- Only supports coordinate pairs (Array)
- Does not take into account the curvature of the earth
- 1 degree of latitude is 66 miles (111km) at the equator
- 1 degree of longitude is 33 miles (55km) in the UK
- Good for small distances and simple

Spherical 2d:

- Full set of GeoJSON objects
- Required for larger areas

There are different methods of utilizing geospatial
Cartesian and Spherical

Geo Indexing Example

$nearSphere

$geoWithin

$geoIntersects

Sortable by Distance

```
> use sample_weatherdata
> db.data.createIndex({"position":"2dsphere"})
> joburg =  {
      "type" : "Point",
"coordinates" : [ -26.2, 28.0 ]
  }

> projection = {position:1,"airTemperature.value":1,_id:0}

> //Distance in metres
> db.data.find({ position: { $nearSphere: { $geometry: joburg,
 $maxDistance: 100000}}}, projection)

{ "position" : { "type" : "Point", "coordinates" : [ -26.1,
28.5 ] }, "airTemperature" : { "value" : 20.5 } }
{ "position" : { "type" : "Point", "coordinates" : [ -26.8,
27.9 ] }, "airTemperature" : { "value" : 19 } }
{ "position" : { "type" : "Point", "coordinates" : [ -26.9,
27.6 ] }, "airTemperature" : { "value" : 20 } }
{ "position" : { "type" : "Point", "coordinates" : [ -27, 27.5
] }, "airTemperature" : { "value" : 20.3 } }
```


$nearSphere, $geoWithin, and $geoIntersects are operators for using geospatial functionality within MongoDB

Exercise - Geo Indexing

Query example: *I would like to stay at "Ribeira Charming Duplex" in Porto but it has no pool - find 5 properties within 5 KM that do*

Write a program (that runs in the shell) that takes the name of a property and finds somewhere nearby with a pool.

```
> use sample_airbnb
> var villaname = "Ribeira Charming Duplex"
> var nearto = db.listingsAndReviews.findOne({name:villaname})
> var position = nearto.address.location
> var query = <write your query here>
> db.listingsAndReviews.find(query,{name:1})
```


Make sure to use the right indexes.

Wildcard Indexes

Dynamic schema makes hard to index all fields - Alternative schemas will be discussed

Wildcard indexes index all fields, or a subtree

Index Entries treat the fieldpath as the first value

Normal index on "user.firstname" index contains "Job","Joe" as keys

Wildcard Index on "user" contains field names in the keys and any other fields in user:

"firstname.Job", "firstname.Joe" , "lastname.Adams", "lastname.Jones"

What does that do to the index size?

What are the performance and hardware implications? This is easy to overuse/misuse. Indexing correctly matters - not just "index everything"

Wildcard indexes are a new feature intended for dynamic schemas
They should not be used as a shortcut to index static schemas.
Even if a single wildcard index could support multiple query fields, MongoDB can use the wildcard index to support only one of the query fields. All remaining fields are resolved without an index.

More Index usage tips

Can use 'hint' to tell MongoDB what Index to use in find

Can use an index for a Regular Expression match,

- If anchored at the start `{ name: /^Joh/ }`
- Beware of the case and that's a range query `>= "Joh"` and `< "Joi"`

Can set a collation order for Indexes (No diacritics, case insensitive, etc.)

- Preferred collation can be specified for collection and views
- Indexes and queries can specify what collation use if not the default

Note: A $regex implementation is not collation-aware.

Indexes in Production

To build an index, the whole data is read. This may be much bigger than the RAM size in the production. So, there were different ways of creating an index:

Foreground Index builds (MongoDB Pre 4.2) - Fast
Required blocking all read-write access to the parent database of the collection being indexed for the duration of the build

Background Index builds (MongoDB Pre 4.2) - Slower
Had less efficient results but allowed read-write access to the database and its collections during the build

Hybrid Index builds (MongoDB 4.2+) - Newer version
Does not lock the server and builds quickly

These were two way of creating index in previous version of MongoDB

In Production: Rolling Build

In production, many maintenance tasks are done in a Rolling Manner

Rolling updates are fast with minimal impact on production.

- Can include creating a new index.
- Change performed on temporarily offline secondary
- Then secondary added back to the replica set
- Secondary catches up using the transaction log

- It starts building an index on the secondary members one at a time, considering it as a standalone member
- It requires at least one replica set election
- Steps to create Rolling Index
    - Remove any one secondary server from the cluster
    - Start secondary as standalone
    - Build the index foreground/hybrid
    - Add secondary back to the cluster with index already created
    - Repeat for all other secondary servers
    - Step down the primary and one of secondary become primary
- Atlas/cloud manager/ops manager do it for you automatically

For workloads that cannot tolerate performance decrease due to index builds, consider using the procedure provided above, to build indexes in a rolling fashion.

In Production: Hidden Indexes

Hidden indexes allow to prevent the DB using an index without dropping it

- Creating an Index is expensive and takes time
- Dropping an Index no longer we think is no needed could be a risk
- Hiding the index temporarily lets us test if it is OK to drop it

Hidden indexes:

- Are no longer used in any operations like find() or update()
- Are still updated and can be re-enabled at any time
- Still apply unique constraints
- If it is a TTL index, documents will still be removed

Hidden indexes are not available before version 4.4

Native text Indexes

Superseded in Atlas by Lucene - but relevant to on-premise

Indexes tokens (words, etc.) used in string fields and allows to search for “contains”

Supports text search for several western languages

Queries are OR by default

Can be compound indexes

Native text Indexes - Algorithm

Algorithm:

- Split text fields into a list of words
- Drop language-specific stop words ("the", "an", "a", "and")
- Apply language-specific suffix stemming ("running", "runs", "runner" all become "run")
- Take the set of stemmed words and make a multikey index

Text indexes use an algorithm to split text fields into words and make them available for search using contains.

Native text Indexes - limits

Logical AND queries can be performed by putting required terms in quotes

- Can be used for required phrases- Example: *"ocean drive"*
- Applies as secondary filter to an OR query for all terms
- Makes AND and Phrase queries very inefficient

No fuzzy matching
Many index entries per document (slow to update)
No wildcard searching
Indexes are smaller than Lucene though

`"ocean drive"` - Ocean OR Drive
`"\"ocean\" \"drive\""` - Ocean AND Drive
`"\"ocean drive\""` - Ocean AND Drive as two consecutive words with one space

Text indexes are limited, and Lucene should be used instead on Atlas

Text Index Example

Only one text index per collection, so create on multiple fields

Index all fields:

```
db.collection.createIndex(
  { "$**": "text" }
)
```

Use $meta and $sort to order if results are small

```
> use sample_airbnb

> db.listingsAndReviews.createIndex({
"reviews.comments": "text",
"summary": "text",
"notes": "text"
})

> db.listingsAndReviews.find({$text : { $search : "dogs"
}})

> db.listingsAndReviews.find({$text:{ $search : "dogs -ca
ts" }})

> db.listingsAndReviews.find({$text : { $search : " \"cof
fee shop\" " }})


> db.listingsAndReviews.find(
  { $text: { $search: "coffee shop cake" } },
   { name: 1, score: { $meta: "textScore" } }
  ).sort( { score: { $meta: "textScore" } } )

//Fails as cannot sort so much data in RAM
```


You can use -term as in "dogs -cats" to say do NOT return results with this term.
You can add a **limit of 10** in the query that fails to sort due to the volume of data in RAM.

Exercise Answers

Answer - Exercise: Indexing

Find the name of the host with the most total listings

```
> db.listingsAndReviews.find({},{"host.host_total_listings_count":1,
"host.host_name":1}).sort({"host.host_total_listings_count":-1}).limit(1)
{ "_id" : "12902610", "host" : { "host_name" : "Sonder", "host_total_listings_count" : 1198 } }
```

Create an index to support the query and show how much more efficient it is:

```
"executionTimeMillis" : 11,
"totalDocsExamined" : 5555,
"works" : 5559,

> db.listingsAndReviews.createIndex({"host.host_total_listings_count": 1})
"executionTimeMillis" : 1,
"totalKeysExamined" : 1,
"totalDocsExamined" : 1,
"works" : 2,
```


Note: In the first query where we apply a sort and limit, MongoDB doesn't guarantee consistent return of the same document especially when there are simultaneous writes happening. It is advisable to include the _id field in sort if consistency is required.

Answer - Exercise: Compound Indexes

Create the best index for this query - how efficient can it be?

```
> query = { amenities: "Waterfront",
  "bed_type" : { $in : [ "Futon", "Real Bed" ] },
  first_review : { $lt: ISODate("2018-12-31") },
  last_review : { $gt : ISODate("2019-02-28") }
}
> project = { bedrooms:1 , price: 1, _id:0, "address.country":1}
> order = {bedrooms:-1,price:1}
> db.listingsAndReviews.find(query,project).sort(order)
//One Answer
> db.listingsAndReviews.createIndex({amenities:1,bedrooms:-1,price:1,
bed_type:1,first_review:1,last_review:1})

DOCS_EXAMINED:             13
KEYS_EXAMINED:             117
TIME:                      0MS
```

Answer - Exercise: Multikey Basics

Simple Arrays : 3 Records
Only 11 Index Entries as only one entry needed for duplicate '3' in third document

`> db.race_results.find( { lap_times : 1 } )` - 1 doc, uses index
`> db.race_results.find( { "lap_times.2" : 3 } )` - 1 doc, doesn't use index but if you add `"lap_time":3` to the query. It partially uses the index to narrow down records

Arrays of Documents: 3 Records,
`> db.blog.find({"comments":{ "name" :"Bob", "rating": 1}})` - 1 doc, uses comments index
`> db.blog.find( { "comments" : { "rating" : 1 } } )` - 0 doc, there is no object of exactly that shape, uses comments index
`> db.blog.find( { "comments.rating" : 1 } )` - 2 docs, using comments.rating index

Answer - Exercise: Multikey Basics

Arrays of Arrays: 5 Documents total

`> db.player.find( { "last_moves" : [ 3, 4 ] } )` - finds 3, using index

`> db.player.find( { "last_moves" : 3 } )` - finds 0 , using index

`> db.player.find( { "last_moves.1" : [ 4, 5 ] } )` - finds 1, does not use index

Using $elemMatch to query:

Cannot index an anonymous value in a multidimensional array, however, it is possible to change the schema to `{ "last_moves" : [ {p:[ 1, 2 ]}, {p:[ 2, 3 ]}, {p:[ 3, 4]} ] }`

Then you could index on `{"lastmoves.p":1}` as a multikey index nested arrays are indexable as long as they are not anonymous.

Answer - Exercise: Geo Indexing

I would like to stay at "Ribeira Charming Duplex" in Porto but it has no pool - find 5 properties within 5 KM that do

Write a single bit of code that may do multiple queries

```
db.listingsAndReviews.createIndex({amenities:1, "address.location":"2dsphere"})
db.listingsAndReviews.createIndex({"name":1})
var villaname = "Ribeira Charming Duplex"
var nearto = db.listingsAndReviews.findOne({name:villaname})
var position = nearto.address.location
query = { "amenities" : "Pool" ,
          "address.location" : { $nearSphere : { $geometry : position, $maxDistance: 5000 }}}

db.listingsAndReviews.find(query,{name:1})
```