DF200

Indexes and Optimization

Optimizing Storage and Retrieval

Release: 20230414

Topics we cover

How do Indexes work

When to use Indexes

How MongoDB uses them

Index types and Options

Performance and limitations

Compression

Bonus: Intro to Time Series Collections

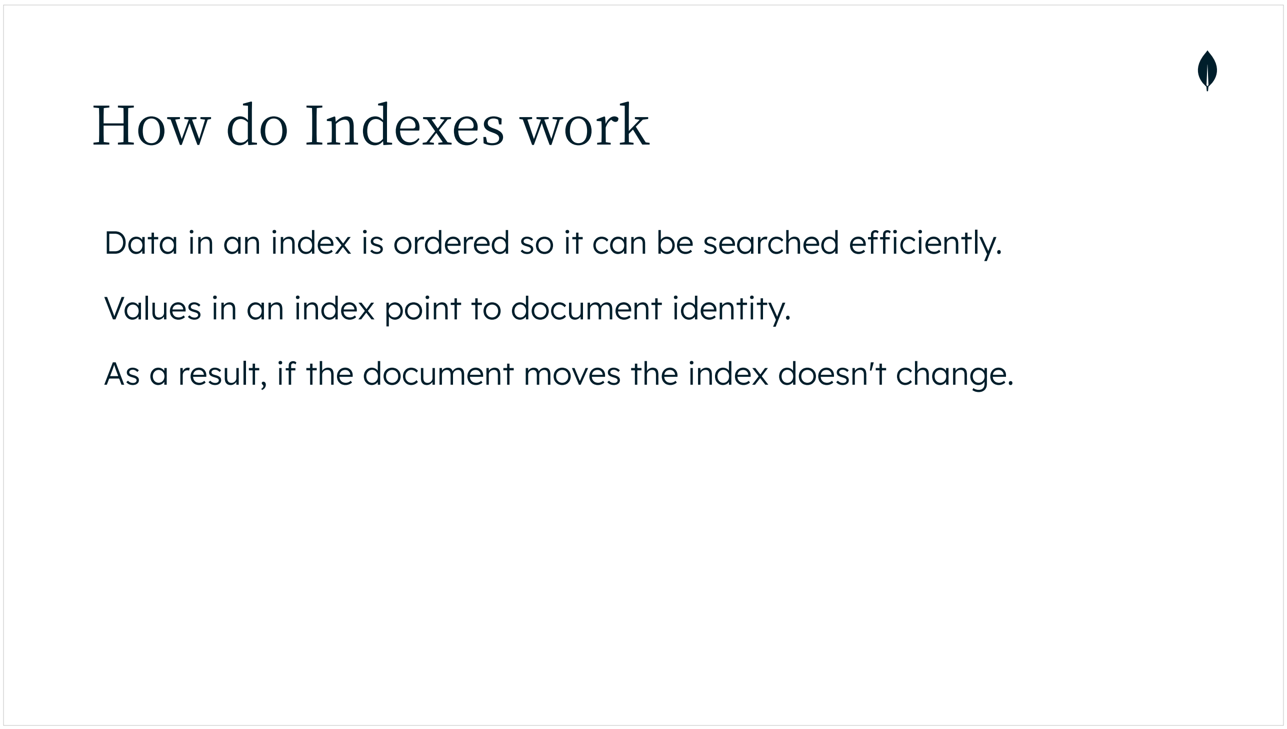
What are Indexes for

Speed up queries and updates

Avoid disk I/O

Reduce overall computation

- The aim is to reduce the resources required for an operation.
- Indexes enable us to dramatically change the resources we use, i.e., memory or disk.
- The disk should be avoided where possible as it is slower than the memory.



How do Indexes work

Data in an index is ordered so it can be searched efficiently.

Values in an index point to document identity.

As a result, if the document moves the index doesn’t change.

- Data in an index is ordered.
  - When looking for specific information, you don't need to look in every document.
  - Indexes are in a binary (BTree) structure, so O(log N) to lookup.
  - Indexes are traversed by value and then point to a document.
- Values in a MongoDB index point to document **identity**.
  - Identity is a 64-bit value in a hidden field assigned when a document is inserted.
  - Documents in collections are in BTrees ordered by **identity**.
  - It's like an internal primary key.
  - It's not maintained between replicas though
  - You can see them by adding .showRecordId() to a cursor
  - If their physical location changes, the index stays the same

Index misconceptions

MongoDB is so fast that it doesn't need indexes

Every field is automatically indexed

NoSQL uses hashes, not indexes

There are some misconceptions about MongoDB indexes:

- MongoDB doesn't use them
- Every field is automatically indexed
- NoSQL uses hashes instead of indexes

When to use an index

A good index should support every query or update

Scanning records is very inefficient, even if it is not all of them

The developer, writing the query, should determine the best index to use

Every query or update should be supported by an index
Developers should be responsible for creating the correct indexing

Index types

Single-field indexes

Compound indexes

Multikey indexes

Geospatial indexes

Text indexes

Hashed indexes

Wildcard indexes

Types of indexes to be covered (single-field, compound, multikey, geospatial, text, wildcard)
Some will be familiar as they are the same as other DB vendors such as Oracle

Using and choosing Indexes

MongoDB checks in the **PlanCache** to see if an optimal index has been chosen before

If not:

- Picks all candidate indexes
- Runs query using them to score which is most efficient
- Adds its choice of best index to the PlanCache

MongoDB uses an empirical technique to look for the best index, this does not require any collection statistics and is well suited
to the simple - single collection type queries mongoDB performs, normally with just one index.

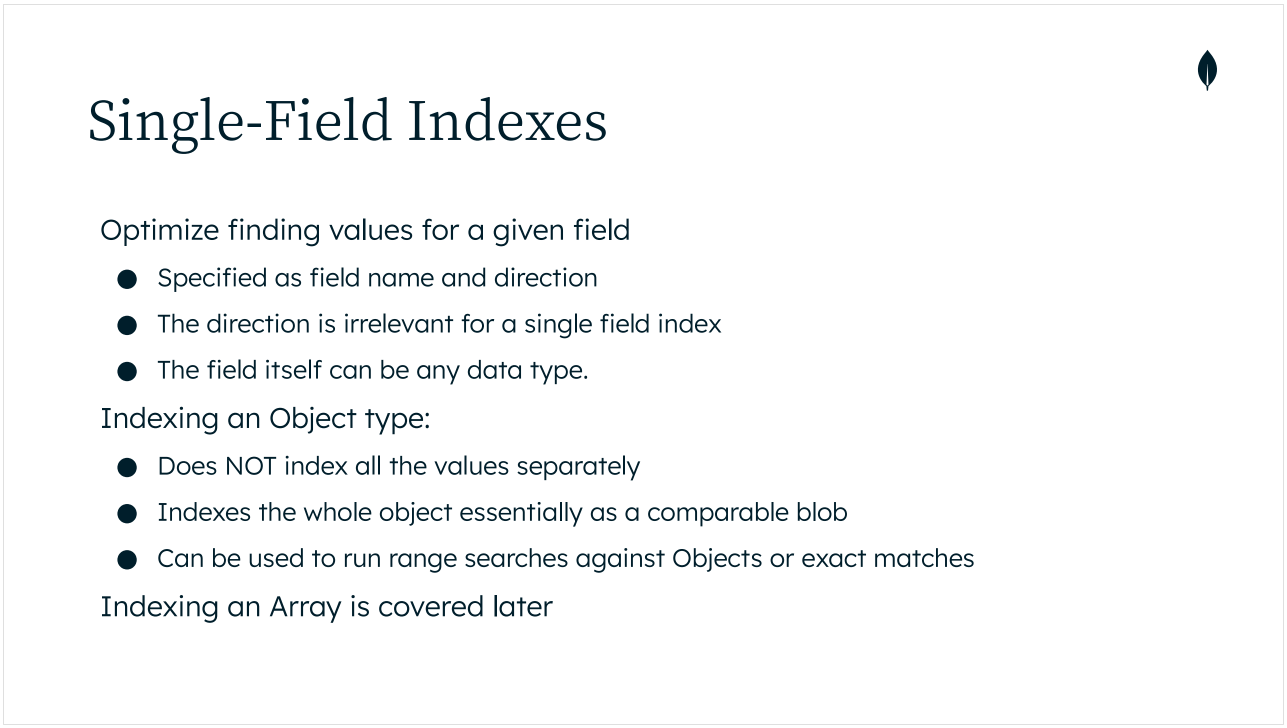
If there are no appropriate indexes then it will run a collection scan.
If there is only one it will use that but test it's performance first.
If there are more than one it could use it will 'Race' them

Using and choosing Indexes

Plan cache entries are evicted when:

- Using that index becomes less efficient
- A new relevant index is added
- The server is restarted

MongoDB keeps a cache of the best index for any given "Shape" of query and how well it performed in the "Race" and it future queries do badly it will have a new "Race"
If new indexes are added that could be used or the server is restarted it will invalidate the cache entry and try again.
We can run commands to see what's in the plan cache if we have a complex issue to debug but normally there will only be one useful index.



Single-Field Indexes

Optimize finding values for a given field

- Specified as field name and direction
- The direction is irrelevant for a single field index
- The field itself can be any data type.

Indexing an Object type:

- Does NOT index all the values separately
- Indexes the whole object essentially as a comparable blob
- Can be used to run range searches against Objects or exact matches

Indexing an Array is covered later

- A single-field index is the most basic kind
- Builds a tree of all values pointing to the documents they are in
- It makes looking up a value O(log n) versus O(n)
- Allows range queries and sorting to be performant too
- Range searching against objects means if we have an objects like {a:1, b:5} with an index we can create a range query to find all values with {a:1} or even {a : {$gt:1}} without knowing b. `{ myObj : { $gte: { a:1 }, $lte : { a:1, b: MaxKey() }}}`
- When querying objects both exact matches and ranges rely on the field order being the same, we must check language does not rearrange the field order.

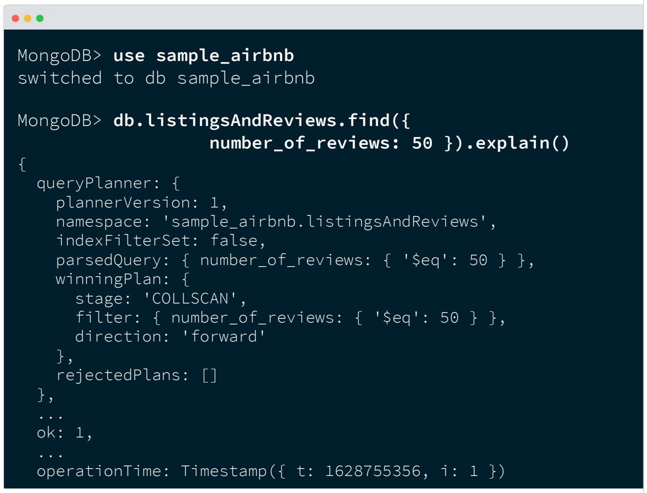
A collection scan

explain() on cursor shows the query mechanics

COLLSCAN: Collection Scan

Every document in the collection looked at

Very Inefficient



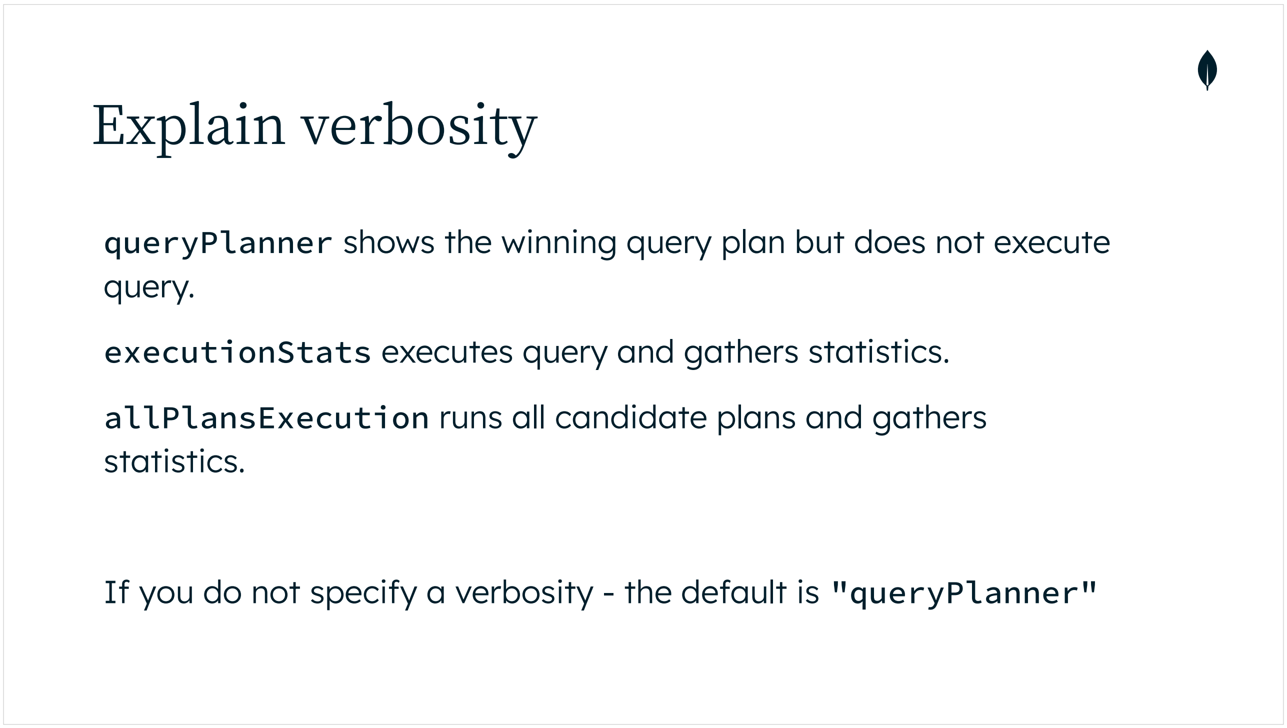
```
MongoDB> use sample_airbnb
switched to db sample_airbnb

MongoDB> db.listingsAndReviews.find({
                    number_of_reviews: 50 }).explain()
{
  queryPlanner: {
    plannerVersion: 1,
    namespace: 'sample_airbnb.listingsAndReviews',
    indexFilterSet: false,
    parsedQuery: { number_of_reviews: { '$eq': 50 } },
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { number_of_reviews: { '$eq': 50 } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  ...
  ok: 1,
  ...
  operationTime: Timestamp({ t: 1628755356, i: 1 })
```

Queries tend to always be quick on small datasets, so we will use tools to see the implications rather than merely observing time.

The example query has no index, so the DB Engine must look through every document in the collection.

If the collection is too big and isn't cached in RAM, then a huge amount of Disk IO will be consumed, and it will be very slow.

Even if in RAM, it is a lot of data to look through



Explain verbosity

`queryPlanner` shows the winning query plan but does not execute query.

`executionStats` executes query and gathers statistics.

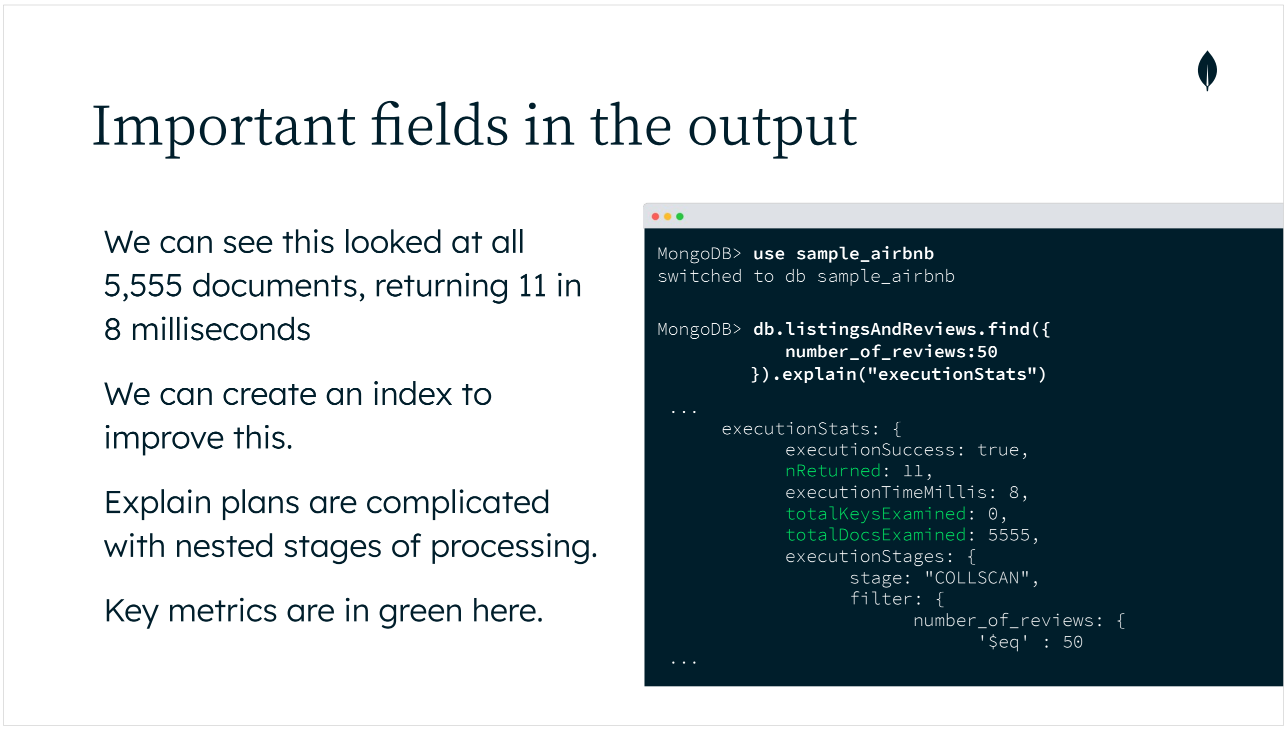
`allPlansExecution` runs all candidate plans and gathers statistics.

If you do not specify a verbosity - the default is `"queryPlanner"`

By default, we only see what the DB Engine intends to do

“executionStats” gives us more detail, such as how long the query takes to run and how much data it has to examine

“allPlansExecution” runs all candidate plans and gathers statistics for comparison



Important fields in the output

We can see this looked at all 5,555 documents, returning 11 in 8 milliseconds

We can create an index to improve this.

Explain plans are complicated with nested stages of processing.

Key metrics are in green here.

```
MongoDB> use sample_airbnb
switched to db sample_airbnb

MongoDB> db.listingsAndReviews.find({
            number_of_reviews:50
         }).explain("executionStats")

 ...
      executionStats: {
            executionSuccess: true,
            nReturned: 11,
            executionTimeMillis: 8,
            totalKeysExamined: 0,
            totalDocsExamined: 5555,
            executionStages: {
                  stage: "COLLSCAN",
                  filter: {
                        number_of_reviews: {
                              '$eq' : 50
 ...
```

- **nReturned** is how many documents this stage returns - e.g., the index may narrow to 100 documents, but then an unindexed filter drops that to 10 documents
- **totalKeysExamined** - number of index entries
- **totalDocsExamined** - number of documents read

Ideally, all three of the above are the same number.

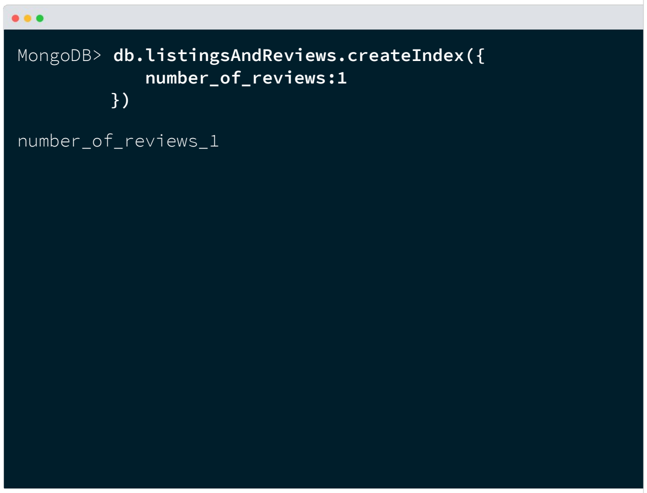
Stage shows us whether a collection scan or index was used.

Creating a simple index

In development, we can instantly create an index using `createIndex()`

In production, we need to look at the impact this will have.

There are better ways to do it in production.



```
MongoDB> db.listingsAndReviews.createIndex({
             number_of_reviews:1
         })

number_of_reviews_1
```

Creating an index takes an object of fields with an ascending index (1) or descending index (-1) option.

In languages where members of objects aren't ordered, the syntax is a little different.

The return value is the name of the newly created index.

Making indexes in the production system needs to consider the impact on the server, cache, disks, and any locking that may occur.

Rolling index builds are generally preferred in clustered production environments.

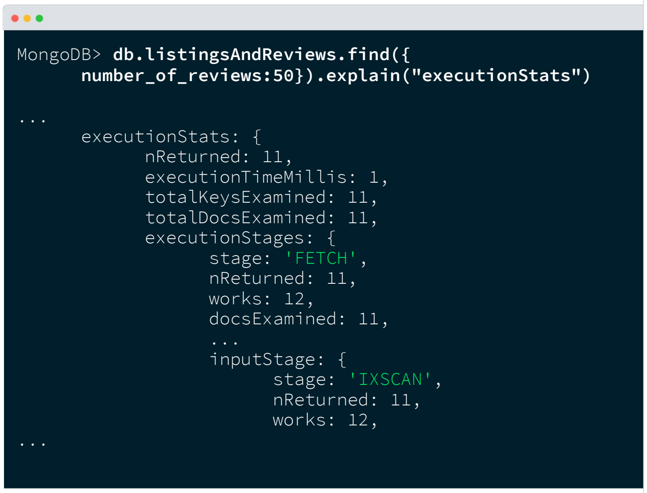
Index Demonstration

Here we see two stages:

IXSCAN (Look through the index) returning a list of 11 documents identities

FETCH (Read known document) getting the document itself

The total time was one millisecond



```
MongoDB> db.listingsAndReviews.find({
      number_of_reviews:50}).explain("executionStats")

...
      executionStats: {
            nReturned: 11,
            executionTimeMillis: 1,
            totalKeysExamined: 11,
            totalDocsExamined: 11,
            executionStages: {
                  stage: 'FETCH',
                  nReturned: 11,
                  works: 12,
                  docsExamined: 11,
                  ...
                  inputStage: {
                        stage: 'IXSCAN',
                        nReturned: 11,
                        works: 12,
...
```

The example shows improved efficiency of using an index - Note how **totalKeysExamined**, **totalDocsExamined**, and **nReturned** are all the same.
Same behavior as an RDBMS

Explainable Operations

find()

aggregate()

count()

update()

remove()

findAndModify()

Explain can be run on the operations shown above
Note that updateOne() or updateMany() does not allow explain, so update() with options can be used

Applying explain()

A flag is sent to the server with the operation to say it's an explain command.

We can set this on the cursor as we do for sort() or limit()

If the function does not return a cursor, the explain flag needs to be set in the collection object instead before calling the function

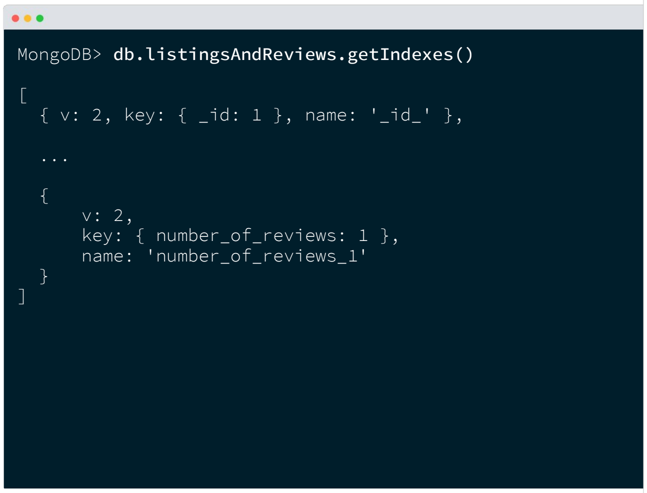
Example: In a count() or update(), we set the flag on the collection object we call it with.

```
peoplecoll = db.people
explainpeoplecoll = peoplecoll.explain()
explainpeoplecoll.count()
```

A query is sent to the server once we start requesting from the cursor - so we set a flag on the cursor to request the explain plan rather than the results.
If we don't have a cursor, calling explain() on a collection returns a collection with that flag set.

Listing Indexes

`getIndexes()` on a collection gives us the index definitions for that collection.

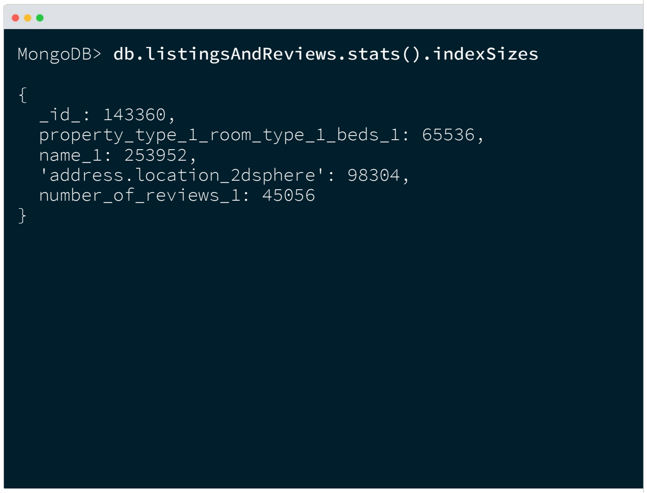


```
MongoDB> db.listingsAndReviews.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },

  ...

  {
      v: 2,
      key: { number_of_reviews: 1 },
      name: 'number_of_reviews_1'
  }
]
```

getIndexes() shows index information (the number of indexes may vary from what you see here on the slide)

Index sizes

We can call `stats()` method and look at the **indexSizes** key to see how large each index is in bytes.



```
MongoDB> db.listingsAndReviews.stats().indexSizes

{
  _id_: 143360,
  property_type_1_room_type_1_beds_1: 65536,
  name_1: 253952,
  'address.location_2dsphere': 98304,
  number_of_reviews_1: 45056
}
```

Among other options, the scaling factor can be passed to see stats in desired unit. **`db.listingsAndReviews.stats({ scale: 1024 }).indexSizes`** shows the index sizes in KB.

Exercise - Indexing

Use the **sample_airbnb** database and the **listingsAndReviews** collection:

Find the name of the host with the most total listings (this is an existing field)

Create an index to support the query

Calculate how much more efficient it is now with this index

Note - this is a tricky question for a number of reasons!

Unique Index

Indexes can enforce a unique constraint

Example,

```
db.a.createIndex({custid: 1}, {unique: true})
```

NULL is a value and so only one record can have a NULL in unique field.

Partial Index

Partial indexes index a subset of documents based on values.

Can greatly reduce index size

Example,

```
db.orders.createIndex(
    { customer: 1 },
    { partialFilterExpression: { archived:
false } } )
```

Partial indexes can be used to only index fields that meet a specified filter expression. Useful for reducing index size.
The Index is only used where the query matches the condition for document to be indexed.

Sparse Index

Sparse Indexes don’t index missing fields.

Example,

```
db.scores.createIndex({score: 1}, {sparse: true})
```

Sparse Indexes are superseded by Partial Indexes

Use `{field: {$exists: true}}` for your partialFilterExpression.

Hashed Indexes

Hashed Indexes index a 20 byte md5 of the BSON value

Support exact match only

Cannot be used for unique constraints

Can potentially reduce index size if original values are large

Downside: random values in a BTree use excessive resources

```
db.people.createIndex({ name : "hashed" })
```

Hashed indexing creates performance challenges for range matches etc., so it should be used with caution.
Random values (Hashes or traditional GUIDS) in a BTree maximize the requirement for RAM and Disk/IO and so should be avoided. This is covered later in the course.

Indexes and Performance

Indexes improve read performance when used

Each index adds **~10%** overhead (Hashed Indexes can add more)

An index is modified any time a document:

- Is inserted (applies to most indexes)
- Is deleted (applies to most indexes)
- Is updated in such a way that its indexed fields change

Indexes must be applied with careful consideration as they do create overhead when writing data
Unused indexes should be identified and removed
At times the partial and sparse indexes remain unaffected while insert or delete operations.

Index Limitations

Up to 64 indexes per collection - Avoid being close to this upper bound

Write performance degrades to unusable between 20 and 30

4 indexes per collection is a good recommended number

The hard limit is 64 indexes per collection, but you should not have anywhere near this number

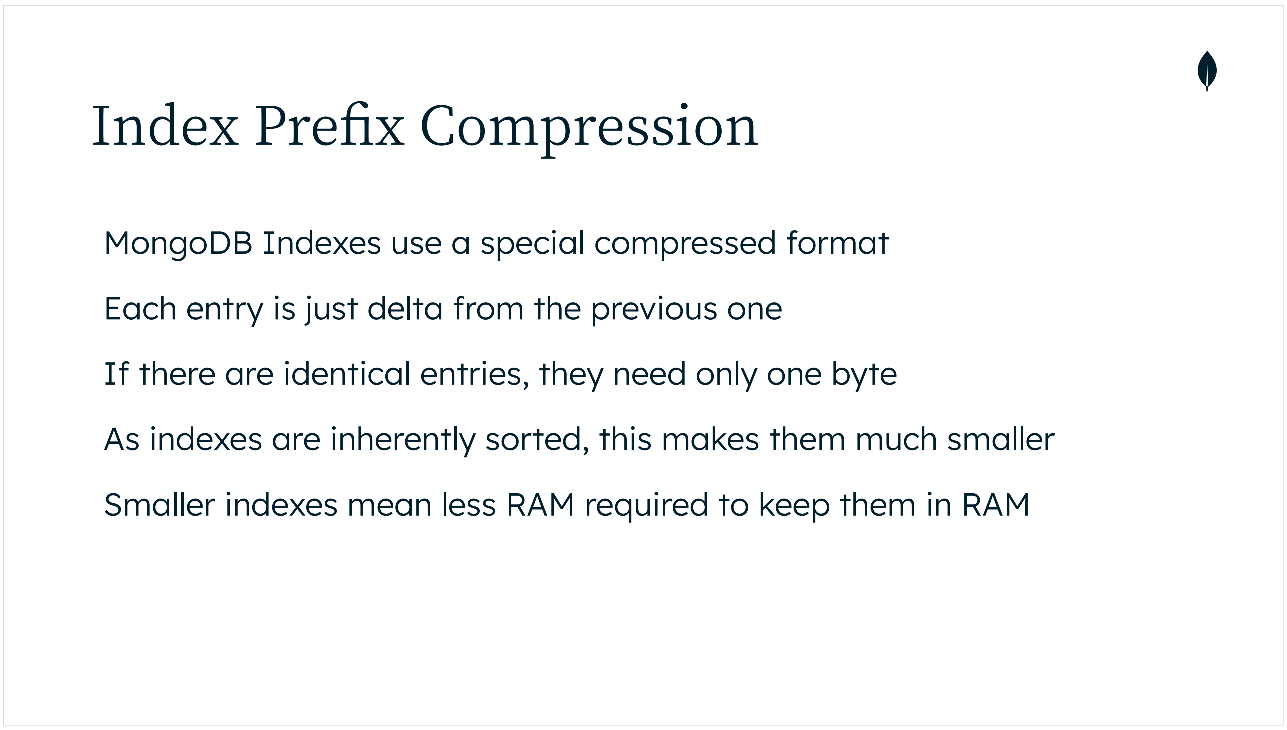
Use Indexes with Care

Every query should use an index

Every index should be used by a query

Indexes require RAM, be mindful about the choice of key

Depending on the size and available resources, indexes will either be used from disk or cached. You should aim to fit indexes in the cache. Otherwise, performance will be seriously impacted.



Index Prefix Compression

MongoDB Indexes use a special compressed format

Each entry is just delta from the previous one

If there are identical entries, they need only one byte

As indexes are inherently sorted, this makes them much smaller

Smaller indexes mean less RAM required to keep them in RAM

MongoDB uses index prefix compression to reduce the space that indexes consume.
Where an entry shares a prefix with a previous entry in the block, it has a pointer to that entry and length, and then the new data.
So subsequent identical keys take very little space. This helps optimize cache usage.

Multikey Indexes

A multikey index is an index that has indexed an array

Can index primitives, documents, or sub-arrays

Are created using `createIndex()` just as like single-field indexes

If any field in the index is ever found to be an **array** then the index is described as a **multikey index**

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

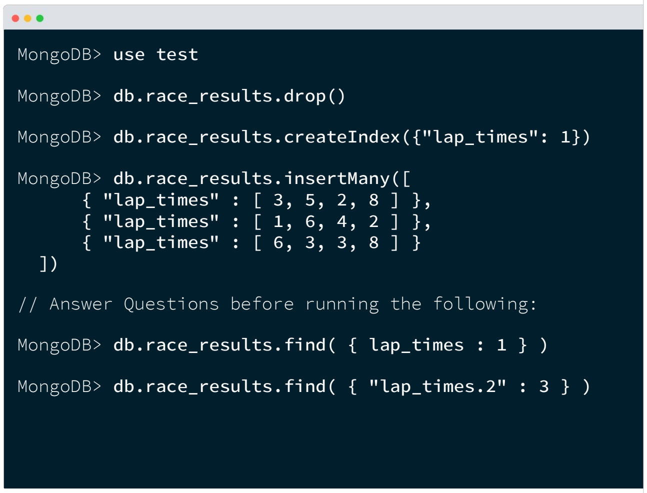
Exercise for the class:

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 } )
```

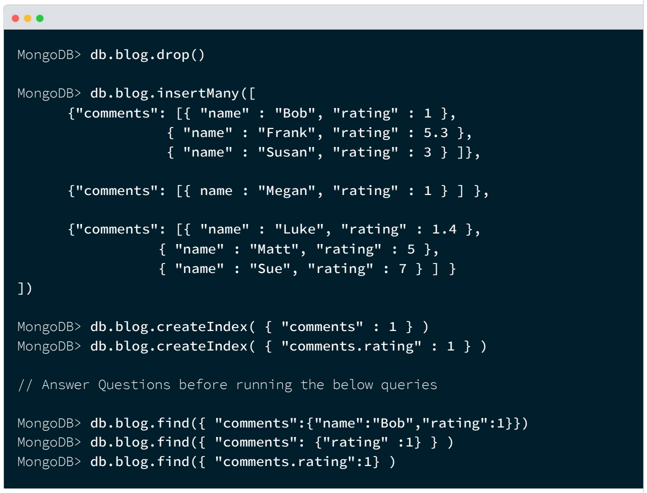
Array of Documents

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} )
```

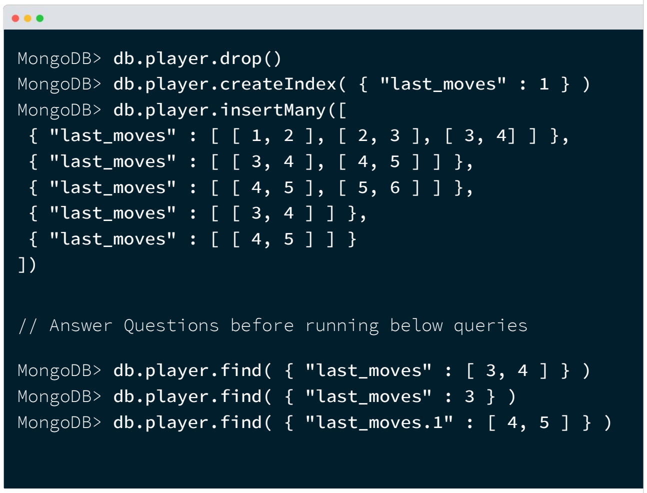
Arrays of Arrays

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 ] } )
```

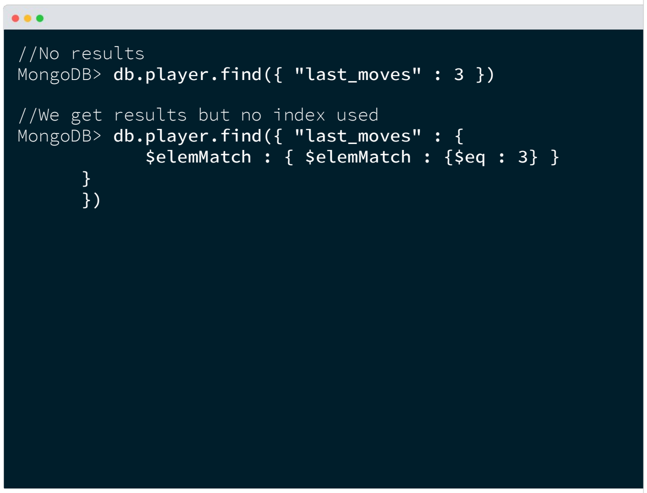
Nested Queries

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?



```
//No results
MongoDB> db.player.find({ "last_moves" : 3 })

//We get results but no index used
MongoDB> db.player.find({ "last_moves" : {
            $elemMatch : { $elemMatch : {$eq : 3} }
      }
      })
```

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.

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

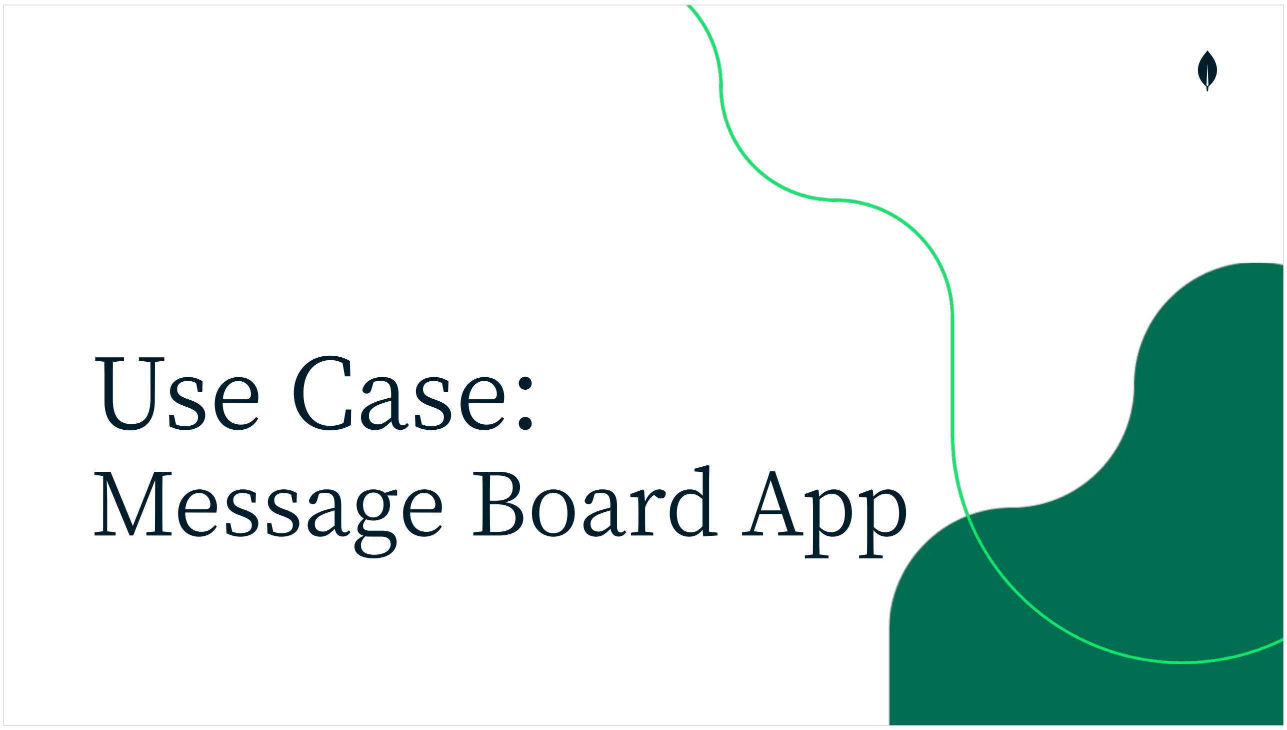
Equality First

- In order of selectivity
- 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 Sort and Range

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

Order of sort should usually be Equality, Sort, Range.
Putting the most selective fields first , can greatly reduce the quantity of index that is in the working set. If you have a field for "Archived" which it true/false having at the start keeps the archived portion of the index out of RAM.
But be aware that Selectivity and Cardinality are different concepts.
https://www.mongodb.com/docs/manual/tutorial/equality-sort-range-rule/



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

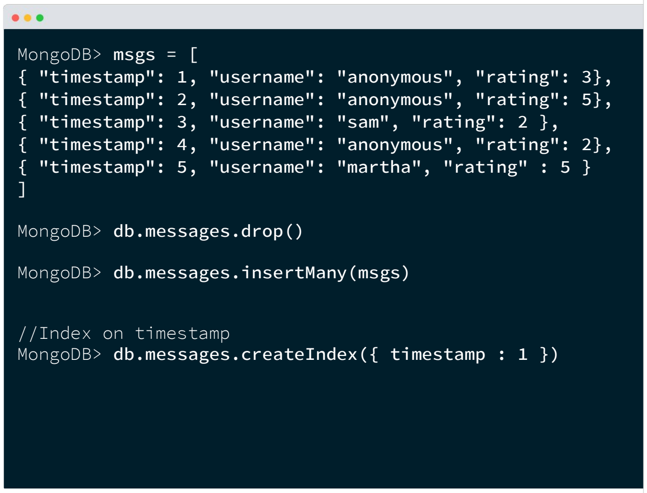
Create five messages (just with the relevant fields):

timestamp

username

rating

Index on the **timestamp**

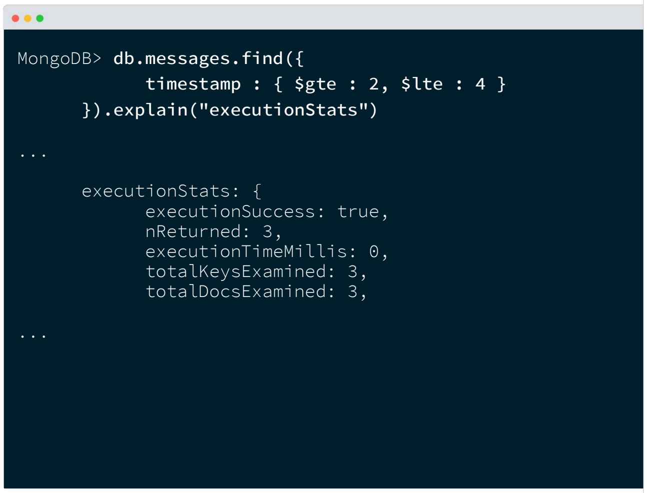


```
MongoDB> msgs = [
{ "timestamp": 1, "username": "anonymous", "rating": 3},
{ "timestamp": 2, "username": "anonymous", "rating": 5},
{ "timestamp": 3, "username": "sam", "rating": 2 },
{ "timestamp": 4, "username": "anonymous", "rating": 2},
{ "timestamp": 5, "username": "martha", "rating" : 5 }
]

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

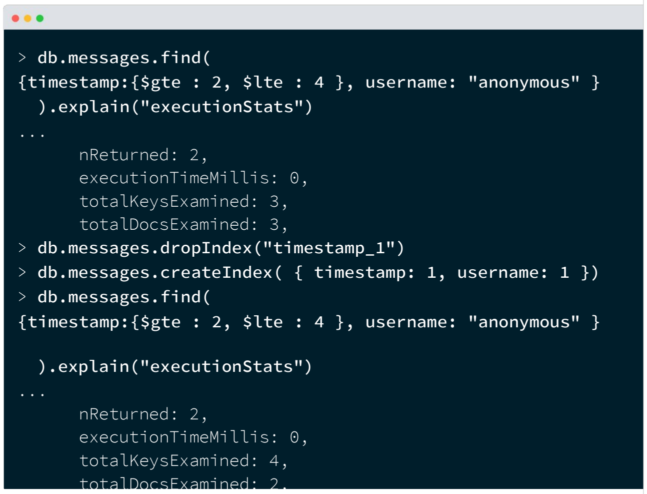
Message Board Index

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.

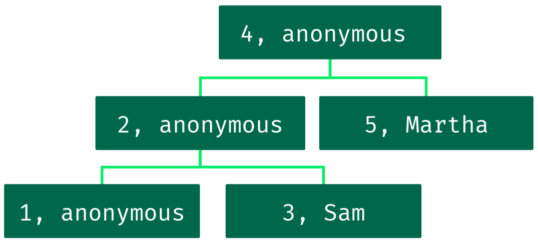
Index fields in wrong order

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

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

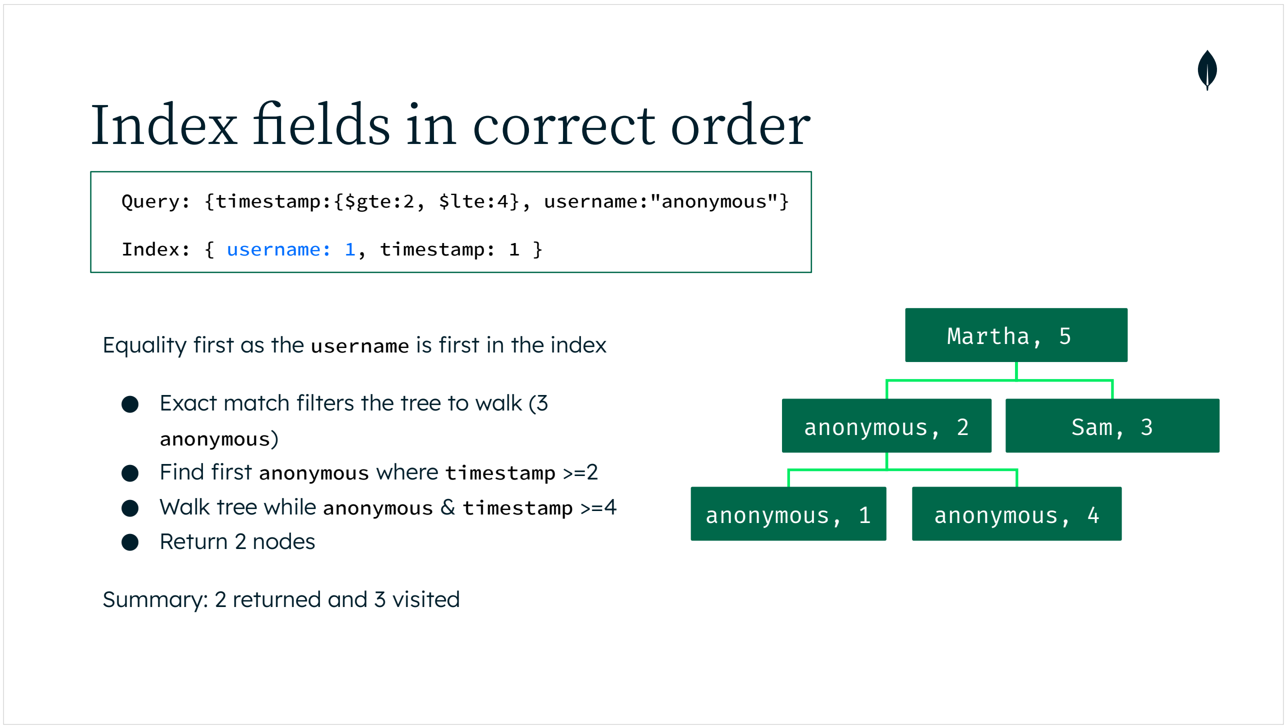
Range first as the `timestamp` is first in the index

- 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.



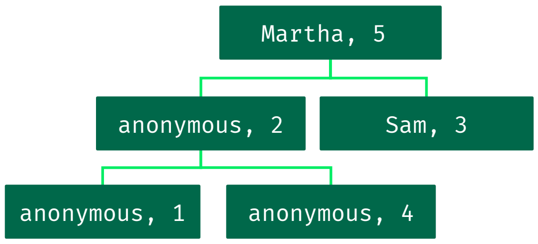
Index fields in correct order

```
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 3 visited

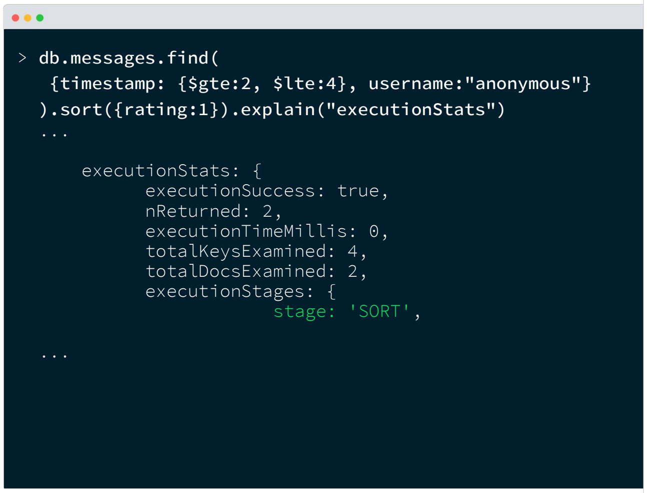


Indexing in the correct order will achieve improved results.

Indexing for Sort

Sort by rating from lowest to highest

Order in the index must match sort order, if not, a reorder is needed in an explicit **SORT** stage



```
> db.messages.find(
   {timestamp: {$gte:2, $lte:4}, username:"anonymous"}
  ).sort({rating:1}).explain("executionStats")
  ...

      executionStats: {
            executionSuccess: true,
            nReturned: 2,
            executionTimeMillis: 0,
            totalKeysExamined: 4,
            totalDocsExamined: 2,
            executionStages: {
                        stage: 'SORT',

  ...
```

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

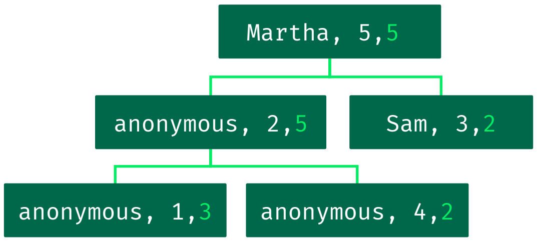
Index in correct order?

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

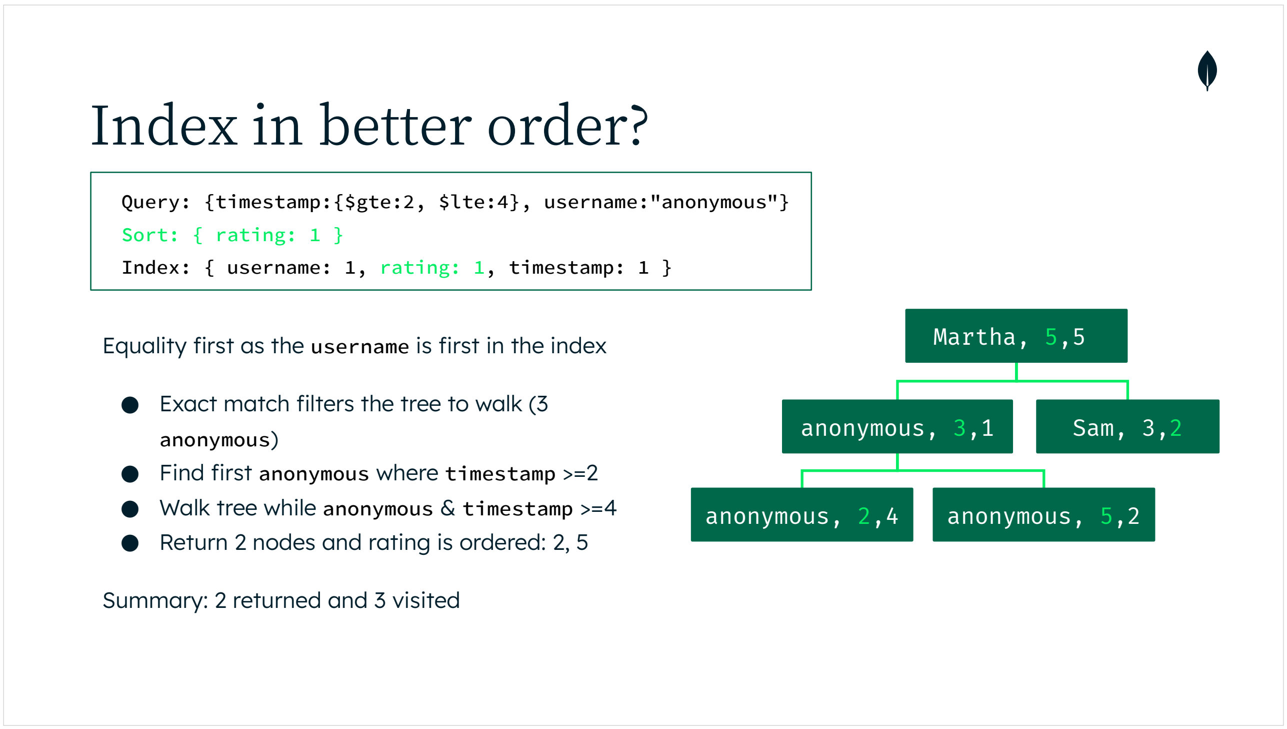
Equality first as the `username` is first in the index

- 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 3 visited + sort stage



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



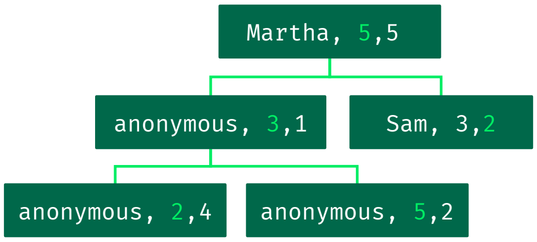
Index in better order?

```
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` where `timestamp` >=2
- Walk tree while `anonymous` & `timestamp` >=4
- Return 2 nodes and rating is ordered: 2, 5

Summary: 2 returned and 3 visited



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

Rules of Compound Indexing

Equality before range

Equality before sorting

Sorting before range

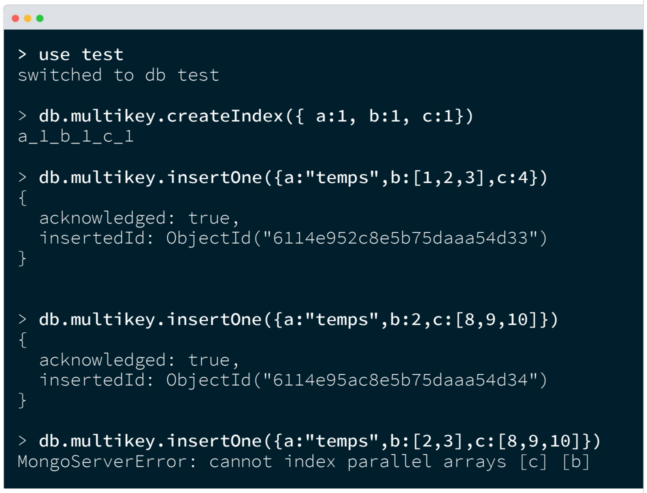
The general rule is Equality, Sort, Range.
Note this is only a generic guideline, there are cases where Range may be better placed before Sort.

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("6114e952c8e5b75daaa54d33")
}


> db.multikey.insertOne({a:"temps",b:2,c:[8,9,10]})
{
  acknowledged: true,
  insertedId: ObjectId("6114e95ac8e5b75daaa54d34")
}

> 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.

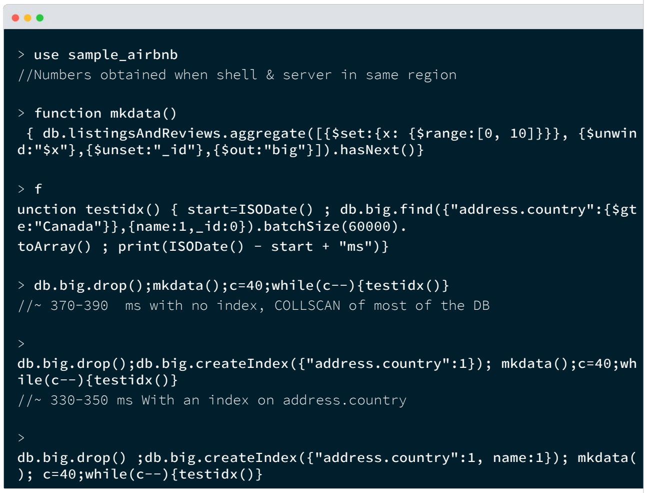
Index covered queries

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 with multikey indexes



```
> use sample_airbnb
//Numbers obtained when shell & server in same region

> function mkdata()
 { db.listingsAndReviews.aggregate([{$set:{x: {$range:[0, 10]}}}, {$unwin
d:"$x"},{$unset:"_id"},{$out:"big"}]).hasNext()}

> f
unction testidx() { start=ISODate() ; db.big.find({"address.country":{$gt
e:"Canada"}},{name:1,_id:0}).batchSize(60000).
toArray() ; print(ISODate() - start + "ms")}

> db.big.drop();mkdata();c=40;while(c--){testidx()}
//~ 370-390  ms with no index, COLLSCAN of most of the DB

>
db.big.drop();db.big.createIndex({"address.country":1}); mkdata();c=40;wh
ile(c--){testidx()}
//~ 330-350 ms With an index on address.country

>
db.big.drop() ;db.big.createIndex({"address.country":1, name:1}); mkdata(
); c=40;while(c--){testidx()}
```

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.

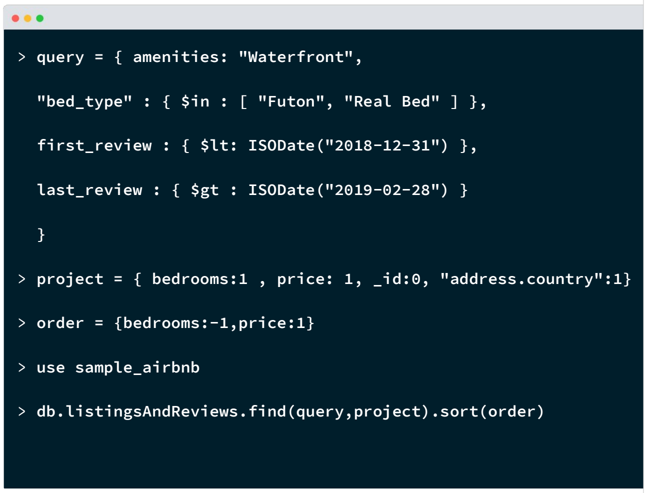
Once we reach our data storage limit we cannot create anything else, including an index - so we are making the index first then making the big collection, this collection technically is larger than our storage limit in the free tier but as it is created in an aggregation we are able to generate it.

toArray() method on a cursor fetches all the contents - we do have some network overhead here though which is a constant in the total time - without that (for example in an aggregation) the difference is much larger proportionally.

Do remember to drop the big collection after the above commands are executed or you will not be able to perform any more write operations.

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)
```

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

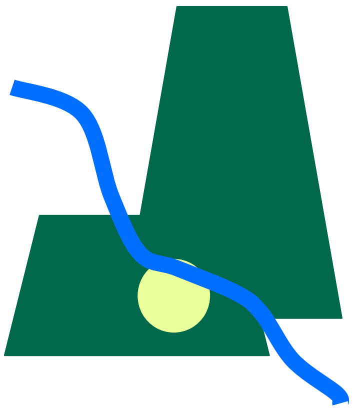
Geospatial Indexing requires a totally different indexing concept.
Geohash transforms 2d coordinate values into 1d key space values, which are then stored in normal MongoDB B-trees.

Geospatial types

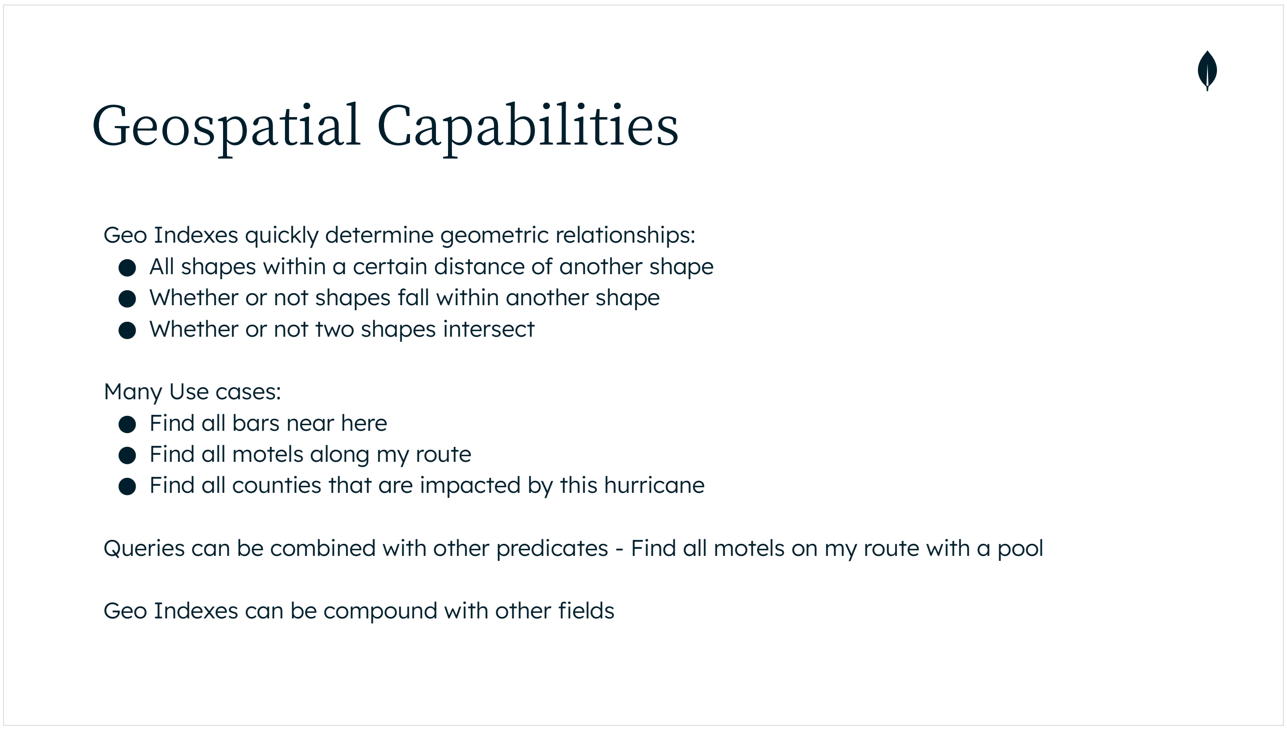
Geospatial data has its own types
- Points
- Lines
- Circles
- Polygons

Groups of the above additive and subtractive

```
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)**

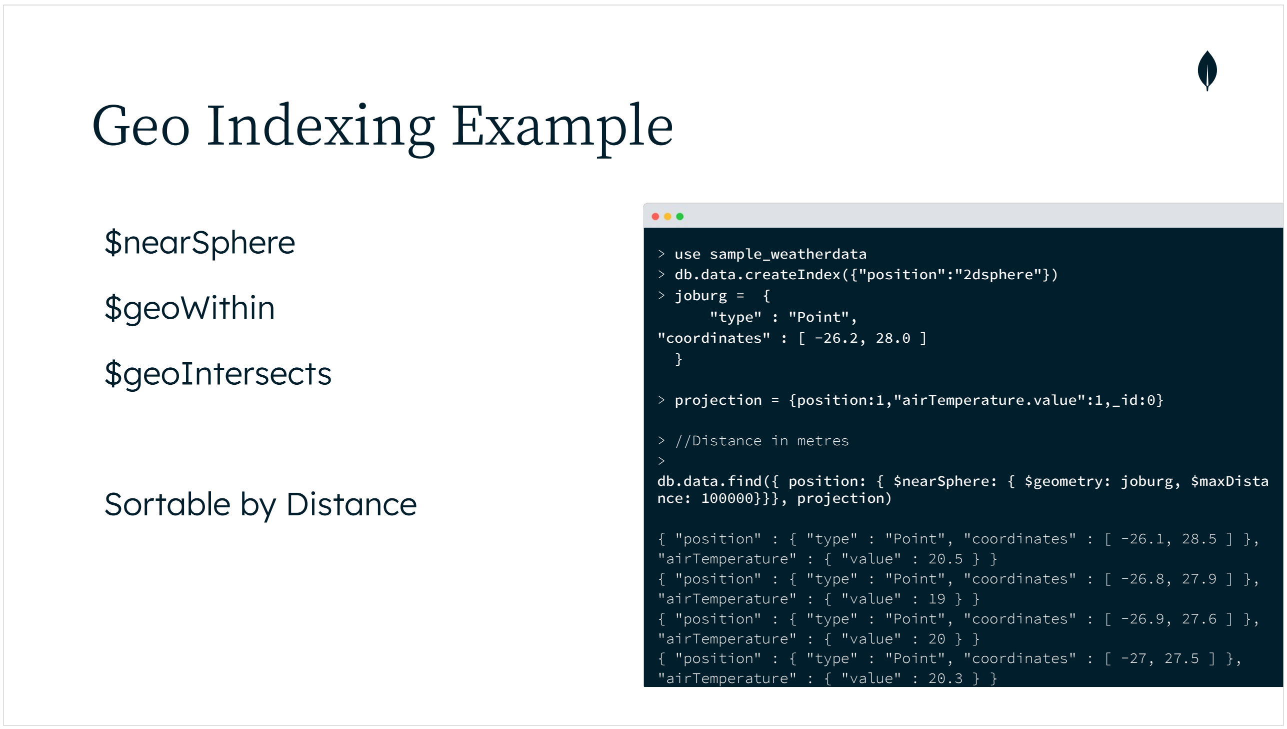
Cartesian:

- 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 longitude 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

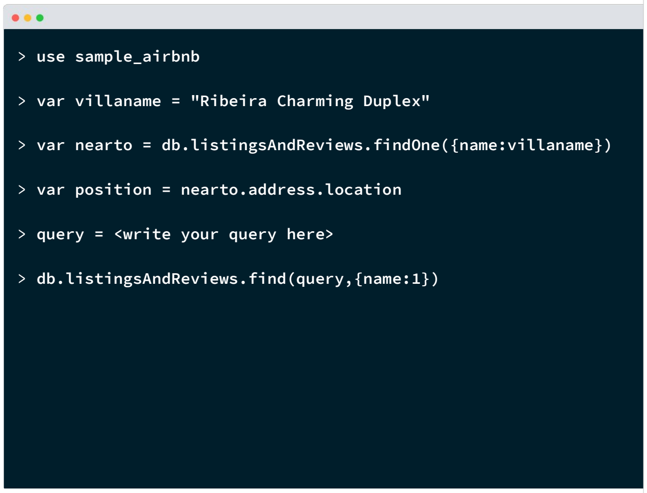
```
> use sample_weatherdata
> db.data.createIndex({"position":"2dsphere"})
> joburg =  {
      "type" : "Point",
"coordinates" : [ -26.2, 28.0 ]
  }

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

> //Distance in metres
>
db.data.find({ position: { $nearSphere: { $geometry: joburg, $maxDista
nce: 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

> query = <write your query here>

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

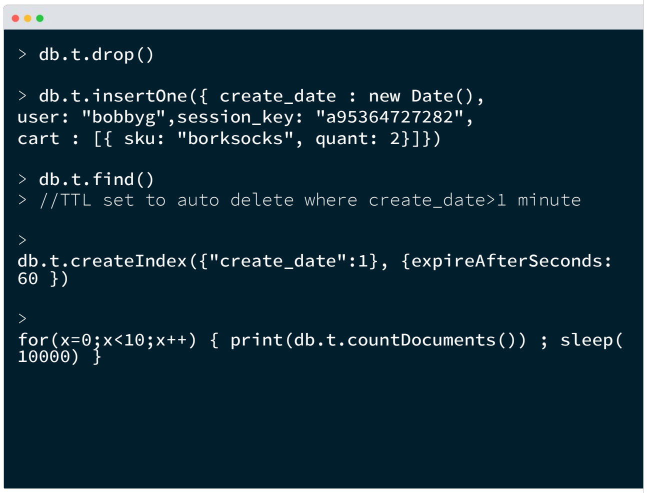
Make sure to use the right indexes.

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()
> //TTL set to auto delete where create_date>1 minute

>
db.t.createIndex({"create_date":1}, {expireAfterSeconds:
60 })

>
for(x=0;x<10;x++) { print(db.t.countDocuments()) ; sleep(
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 (expire-on)
- 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.

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

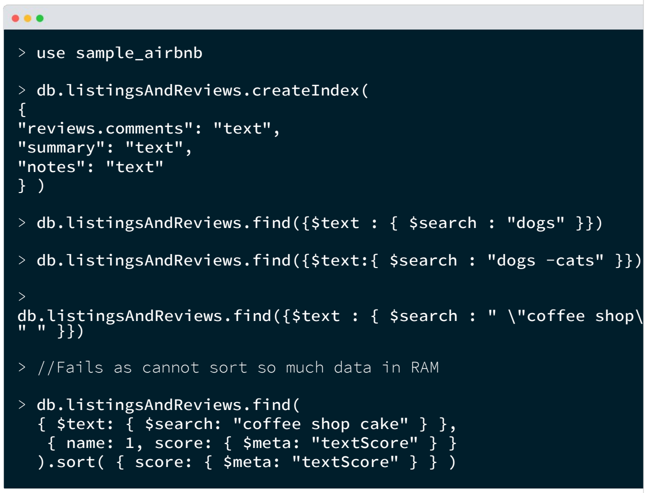
Text Index Example

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

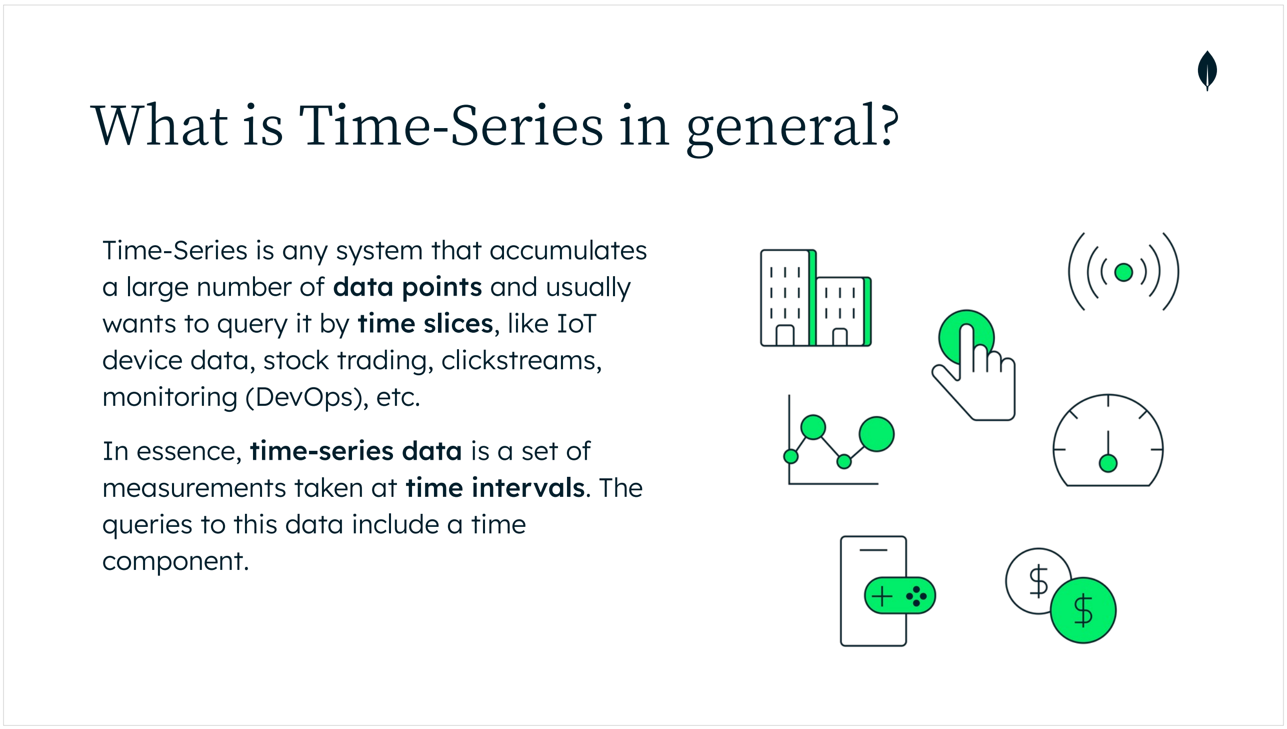


```
> use sample_airbnb

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

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

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

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

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

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

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.

Bonus: Intro to Time Series Collections

Time Series Collections are not directly related to Indexing but they help us optimize storage and retrieval of data that specifically deals with time (or time series). They use indexing in the background and also require some basic knowledge of a bucketting pattern. We will cover all the necessary topics with a view to understanding Time Series Collections better.

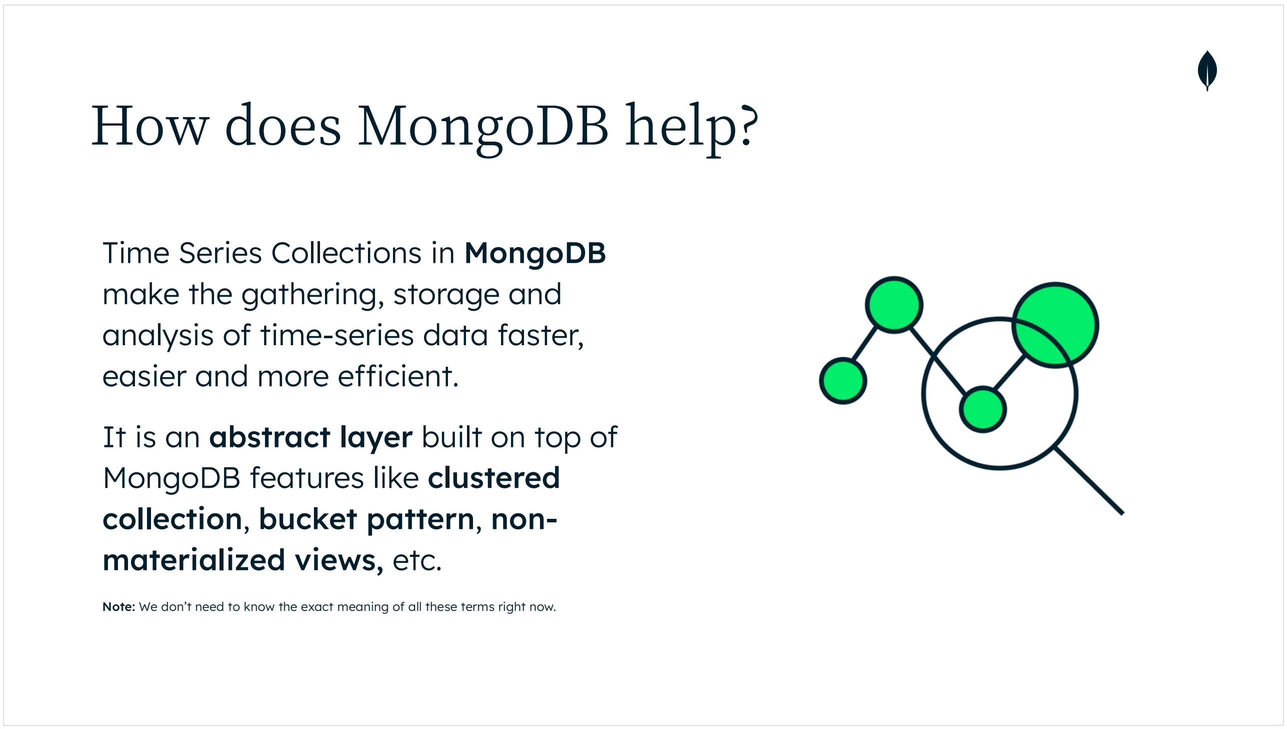


What is Time-Series in general?

Time-Series is any system that accumulates a large number of **data points** and usually wants to query it by **time slices**, like IoT device data, stock trading, clickstreams, monitoring (DevOps), etc.

In essence, **time-series data** is a set of measurements taken at **time intervals**. The queries to this data include a time component.

The Key is that we aren't looking at one record but at a number of them covering a specific time period. Time series is about looking at how data Changes rather than a snapshot of data.
There is a growing popularity of such kind of systems and hence a demand of storing such data efficiently.

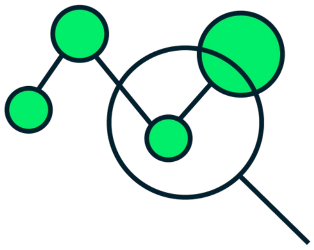


How does MongoDB help?

Time Series Collections in **MongoDB** make the gathering, storage and analysis of time-series data faster, easier and more efficient.

It is an **abstract layer** built on top of MongoDB features like **clustered collection**, **bucket pattern**, **non-materialized views,** etc.

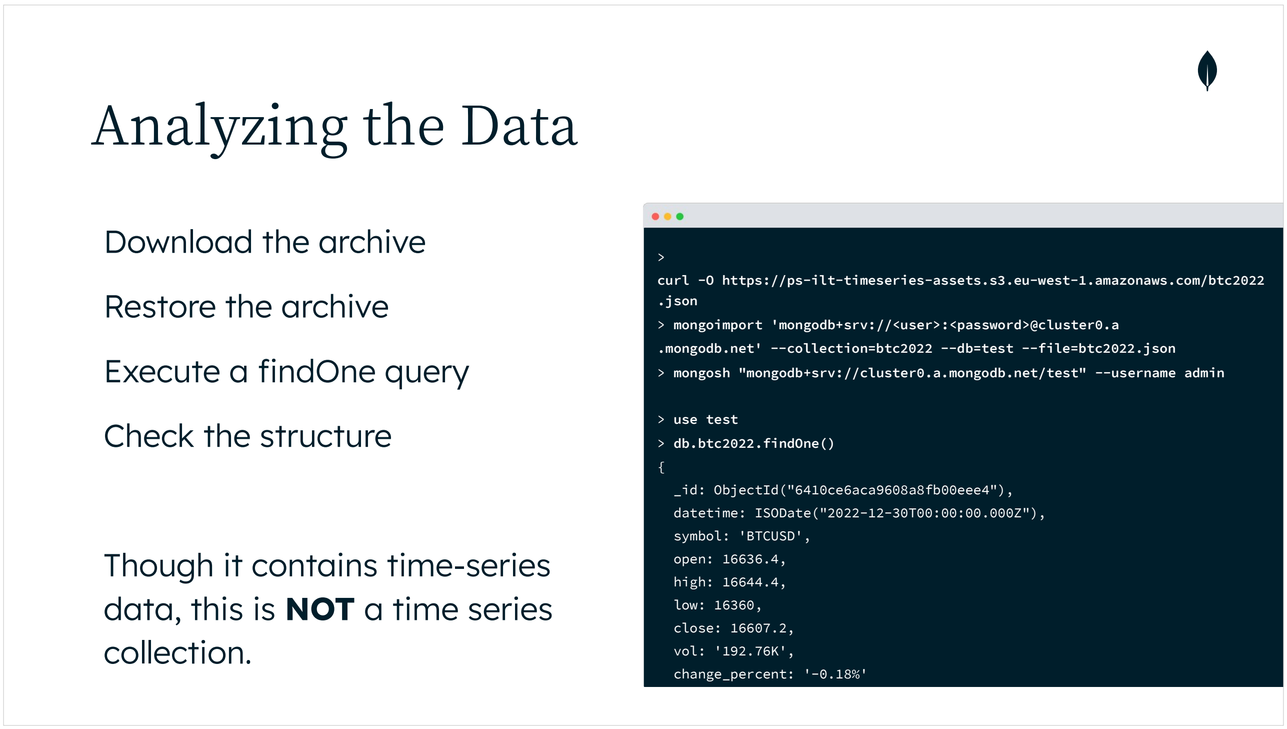
Note: We don't need to know the exact meaning of all these terms right now.



Note: One doesn't need to know every aspect of the internals of time series to start learning (and even implementing) Time Series Collections. We would be covering the required topics as and when necessary.

In MongoDB normally each document is stored separately and if one wants to co-locate data, the developer has to generate these combined records.
In Time series - additional info, when creating the collection, allows the database to reorganize data behind the scenes, to co-locate relevant data, and apply additional compression and indexing allowing it to be retrieved faster.

Use case:
BTC Ticker



Analyzing the Data

Download the archive

Restore the archive

Execute a findOne query

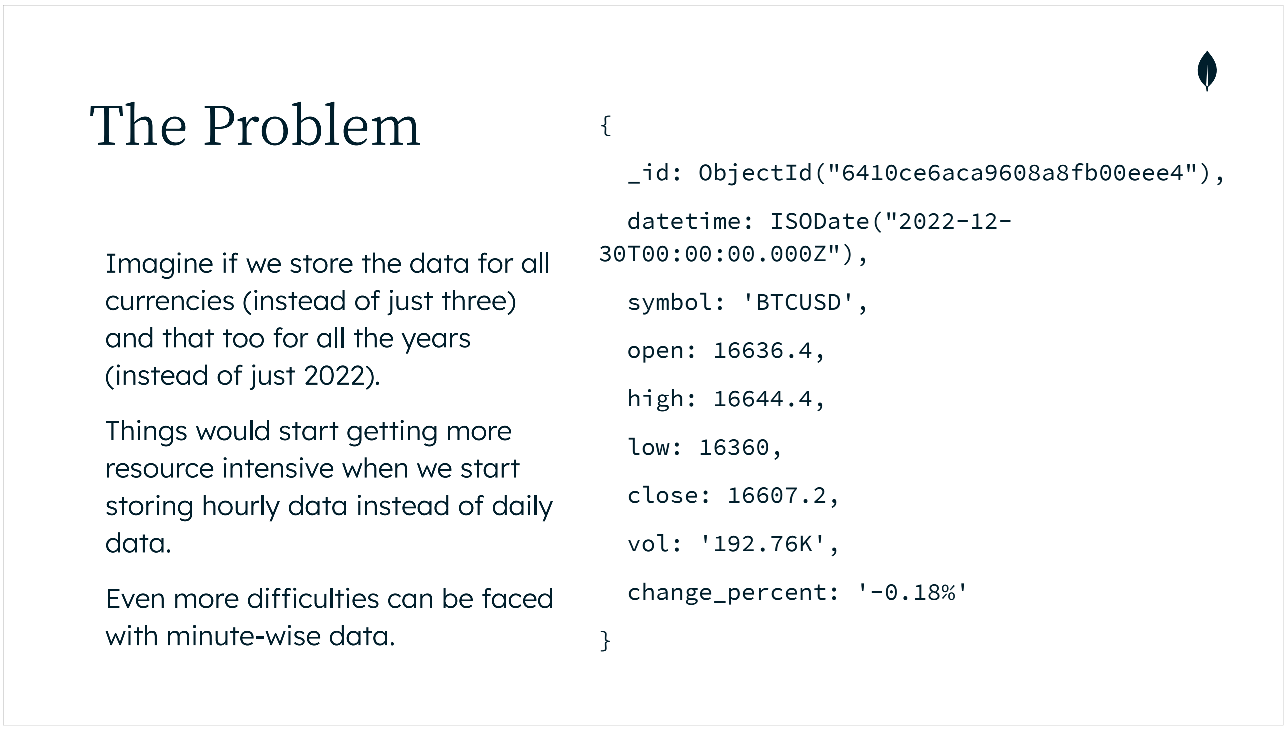
Check the structure

Though it contains time-series data, this is **NOT** a time series collection.

```
>
curl -O https://ps-ilt-timeseries-assets.s3.eu-west-1.amazonaws.com/btc2022
.json
> mongoimport 'mongodb+srv://<user>:<password>@cluster0.a
.mongodb.net' --collection=btc2022 --db=test --file=btc2022.json
> mongosh "mongodb+srv://cluster0.a.mongodb.net/test" --username admin

> use test
> db.btc2022.findOne()
{
  _id: ObjectId("6410ce6aca9608a8fb00eee4"),
  datetime: ISODate("2022-12-30T00:00:00.000Z"),
  symbol: 'BTCUSD',
  open: 16636.4,
  high: 16644.4,
  low: 16360,
  close: 16607.2,
  vol: '192.76K',
  change_percent: '-0.18%'
```

This document shows us the crypto market related data of BitCoin vs USD. The fields open, high, low, close represent the opening price, highest price, lowest price and closing price respectively of one BitCoin on that day (essentially a candlestick). The entire btc2022 collection consists of the data of symbols BTC**USD**, BTC**GBP** and BTC**AUD** for the year 2022 only. This amounts to roughly 1K documents.

The collection btc2022 is just used for data storage purposes, so that we can insert data into the actual time series collection later.



The Problem

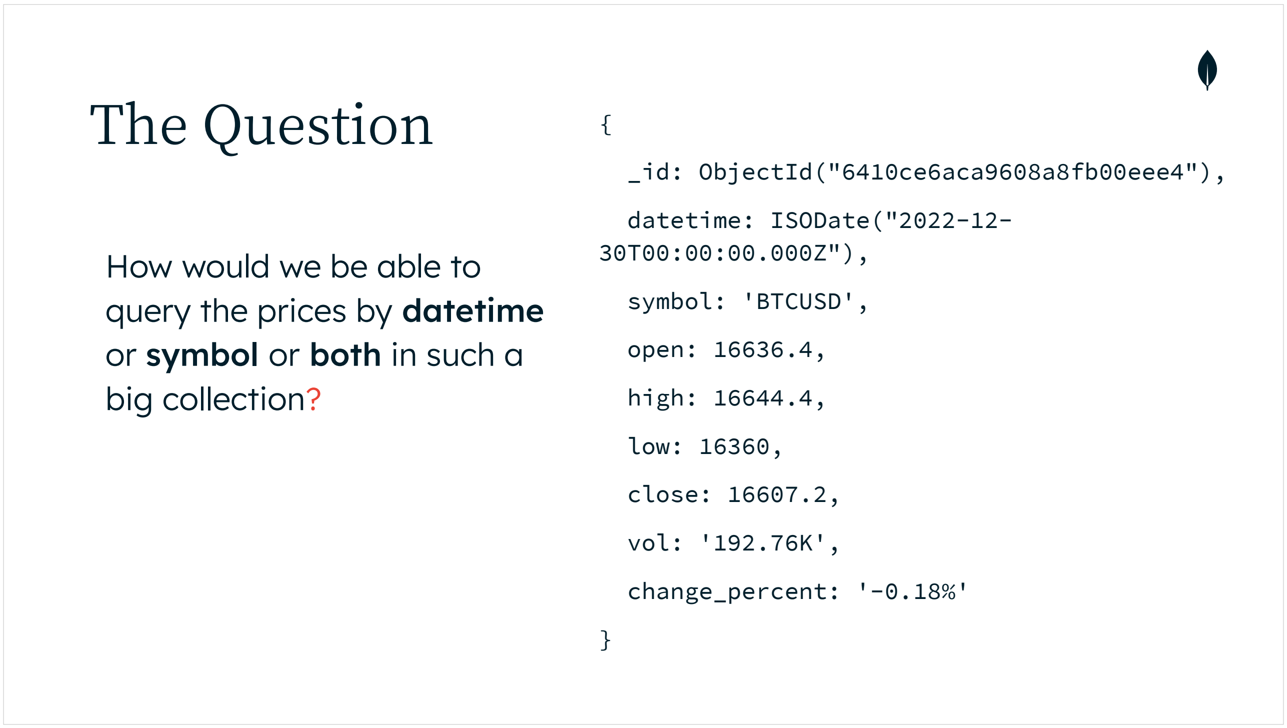
Imagine if we store the data for all currencies (instead of just three) and that too for all the years (instead of just 2022).

Things would start getting more resource intensive when we start storing hourly data instead of daily data.

Even more difficulties can be faced with minute-wise data.

```
{
  _id: ObjectId("6410ce6aca9608a8fb00eee4"),
  datetime: ISODate("2022-12-
30T00:00:00.000Z"),
  symbol: 'BTCUSD',
  open: 16636.4,
  high: 16644.4,
  low: 16360,
  close: 16607.2,
  vol: '192.76K',
  change_percent: '-0.18%'
}
```

Sometimes time-series data will come into your database at high frequency - use-cases like financial transactions, stock market data, readings from smart meters, or metrics from services you're hosting over hundreds or even thousands of servers. In other cases, each measurement may only come in every few minutes. Maybe you're tracking the number of servers that you're running every few minutes to estimate your server costs for the month. Perhaps you're measuring the soil moisture of your favourite plant once a day.

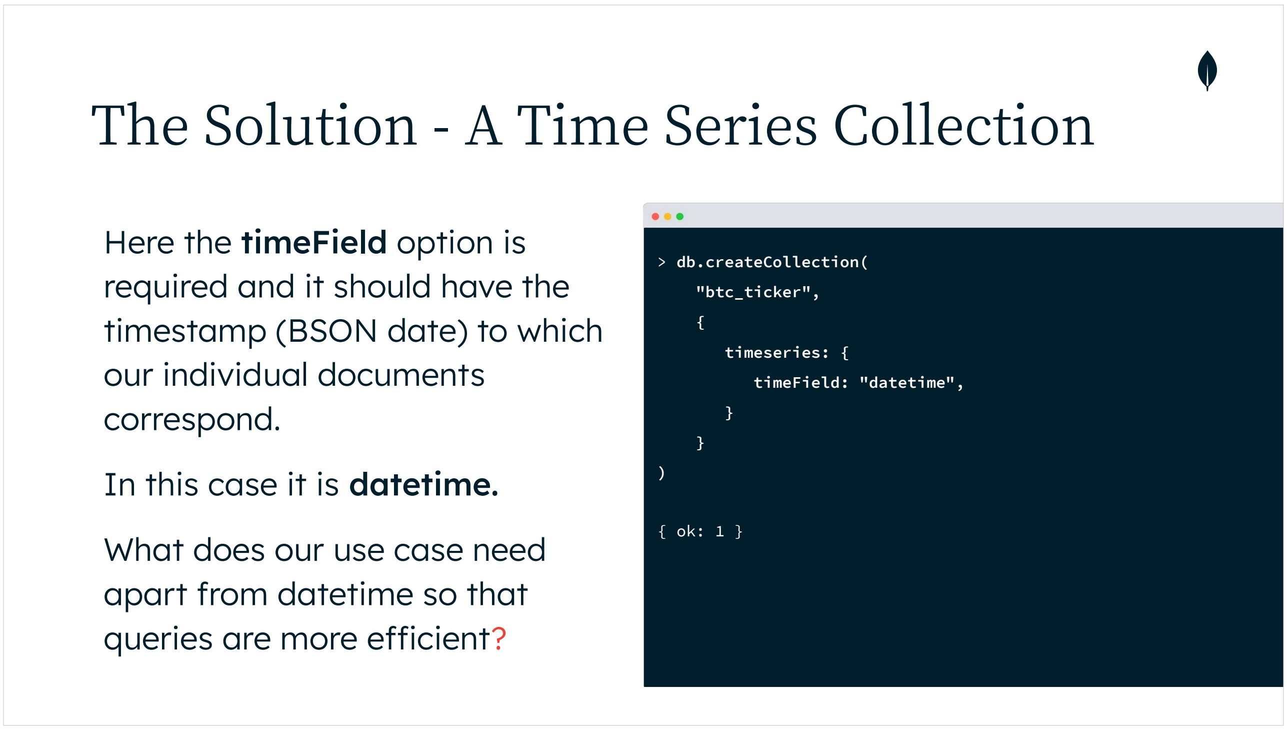


The Question

How would we be able to query the prices by **datetime** or **symbol** or **both** in such a big collection?

```
{
  _id: ObjectId("6410ce6aca9608a8fb00eee4"),
  datetime: ISODate("2022-12-
30T00:00:00.000Z"),
  symbol: 'BTCUSD',
  open: 16636.4,
  high: 16644.4,
  low: 16360,
  close: 16607.2,
  vol: '192.76K',
  change_percent: '-0.18%'
}
```

We can create a compound index, but then, imagine the size of the index.
The primary concern of various MongoDB users who use indexing rather than bucketing is that the indexes get very large in size for small documents which are plenty in volume.
It requires a more sophisticated solution than merely indexing.



The Solution - A Time Series Collection

Here the **timeField** option is required and it should have the timestamp (BSON date) to which our individual documents correspond.

In this case it is **datetime.**

What does our use case need apart from datetime so that queries are more efficient?

```
> db.createCollection(
    "btc_ticker",
    {
       timeseries: {
          timeField: "datetime",
       }
    }
)

{ ok: 1 }
```

Though this is enough to create a time series collection, our case requires the data to be categorized by symbol name.

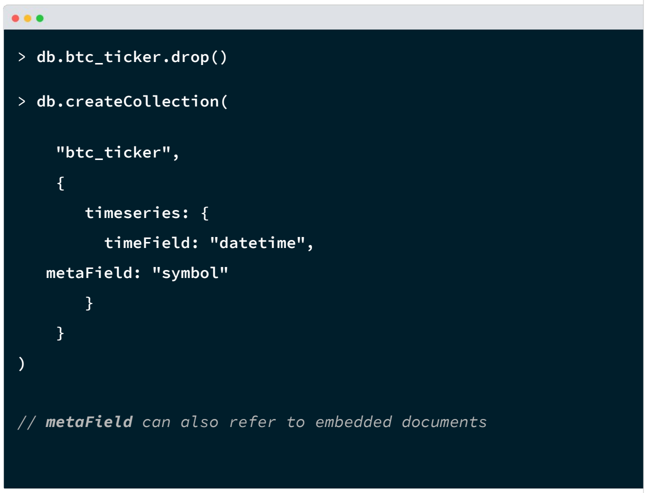
Follow Along Exercise - the metaField

We need to **label** our price data with a **symbol**

In our case, symbol uniquely identifies the price on a particular **datetime**.

metaField should be the one that doesn’t change or rarely changes.

metaField is virtually a **secondary index** in most of the use cases.



```
> db.btc_ticker.drop()

> db.createCollection(

    "btc_ticker",
    {
       timeseries: {
         timeField: "datetime",
   metaField: "symbol"
       }
    }
)

// metaField can also refer to embedded documents
```

metaField can also refer to an embedded document in the source data. For instance **`metaData:{symbol:'BTCUSD', exchange:'COINBASE'}`**

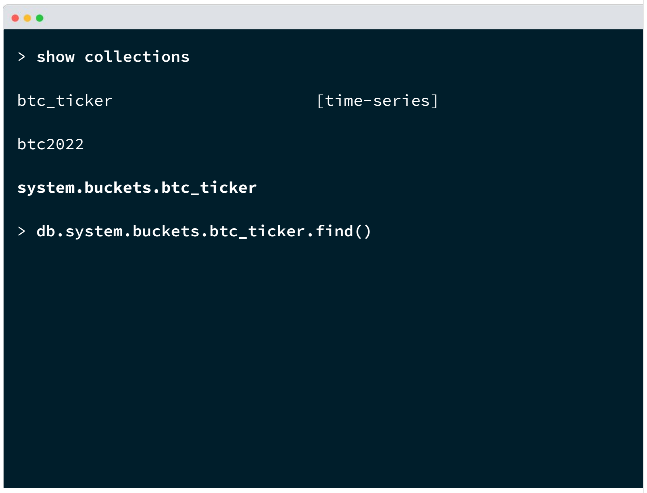
Three objects get created

The `btc_ticker` collection which is actually a **writable view**.

`systems.buckets.btc_ticker` where the **actual data** is stored.

`system.views` mapping the time series collection with its underlying bucket collection.

Do you think that `systems.buckets.btc_ticker` would have any data?



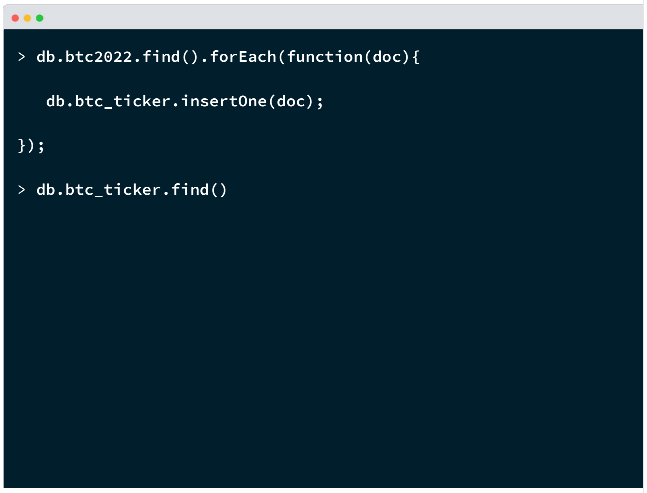
```
> show collections

btc_ticker                              [time-series]

btc2022

system.buckets.btc_ticker

> db.system.buckets.btc_ticker.find()
```

Follow Along Exercise - Inserting data

Insert operation is done as usual.

What output would you expect from the **`btc_ticker`** collection?

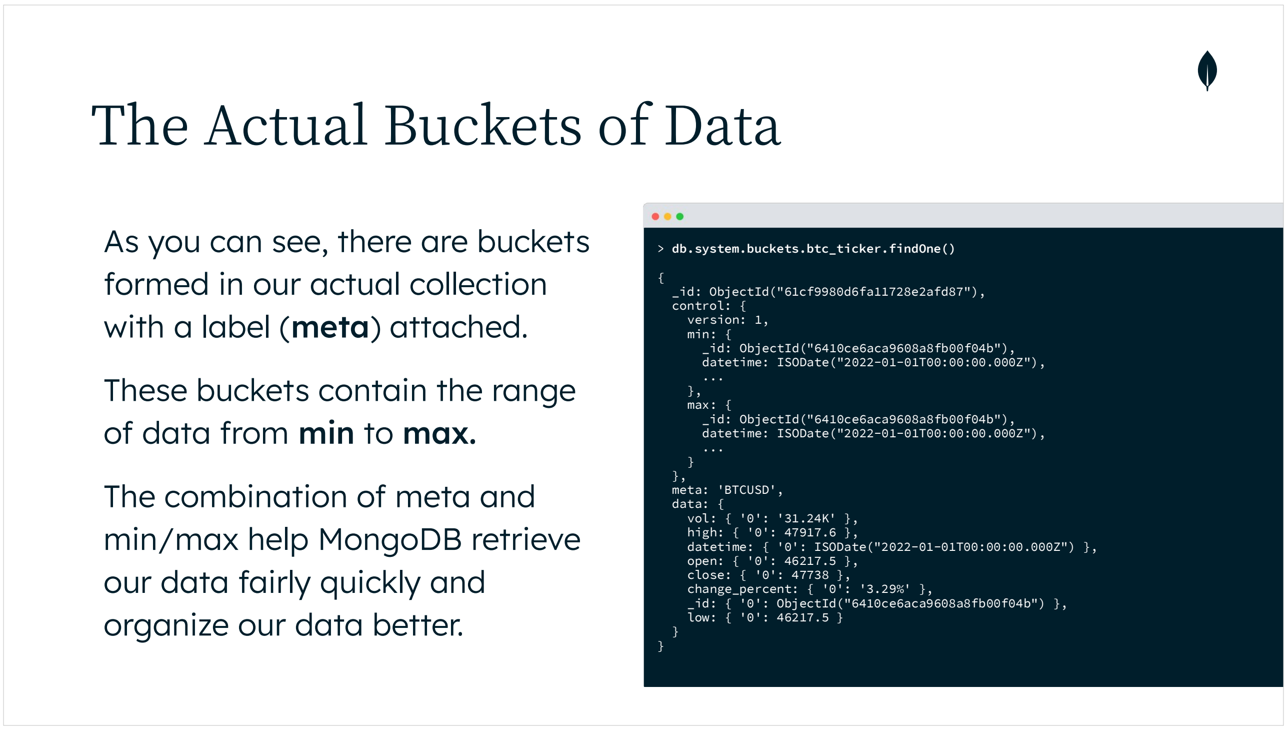
Which collection would be more interesting to check at this point?



```
> db.btc2022.find().forEach(function(doc){

   db.btc_ticker.insertOne(doc);

});

> db.btc_ticker.find()
```

In this case we are just inserting data from btc2022 collection that we imported earlier. In real-world, the data might come from an application in real time.

The documents should be inserted in time order (due to the nature of use cases in which these collections are used) usually and that would be a better structure for time series collection.



The Actual Buckets of Data

As you can see, there are buckets formed in our actual collection with a label (**meta**) attached.

These buckets contain the range of data from **min** to **max.**

The combination of meta and min/max help MongoDB retrieve our data fairly quickly and organize our data better.

```
> db.system.buckets.btc_ticker.findOne()

{
  _id: ObjectId("61cf9980d6fa11728e2afd87"),
  control: {
    version: 1,
    min: {
      _id: ObjectId("6410ce6aca9608a8fb00f04b"),
      datetime: ISODate("2022-01-01T00:00:00.000Z"),
      ...
    },
    max: {
      _id: ObjectId("6410ce6aca9608a8fb00f04b"),
      datetime: ISODate("2022-01-01T00:00:00.000Z"),
      ...
    }
  },
  meta: 'BTCUSD',
  data: {
    vol: { '0': '31.24K' },
    high: { '0': 47917.6 },
    datetime: { '0': ISODate("2022-01-01T00:00:00.000Z") },
    open: { '0': 46217.5 },
    close: { '0': 47738 },
    change_percent: { '0': '3.29%' },
    _id: { '0': ObjectId("6410ce6aca9608a8fb00f04b") },
    low: { '0': 46217.5 }
  }
}
```

The entire complex process of showing the data in our view from the buckets of data is carried out by MongoDB internally.

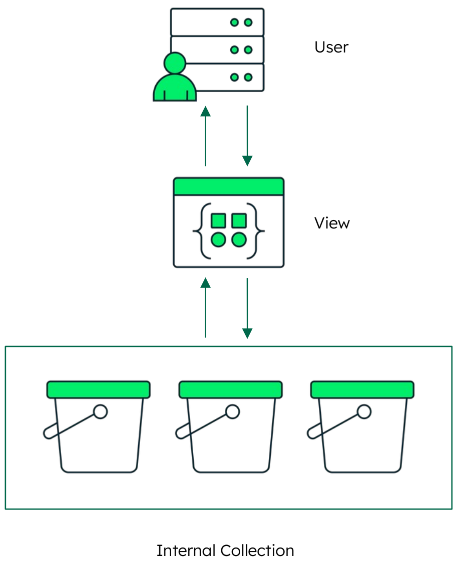
The formal definition

A time-series collection is an abstract layer containing a writable **non-materialized view** on top of an **internal (bucket) collection**.

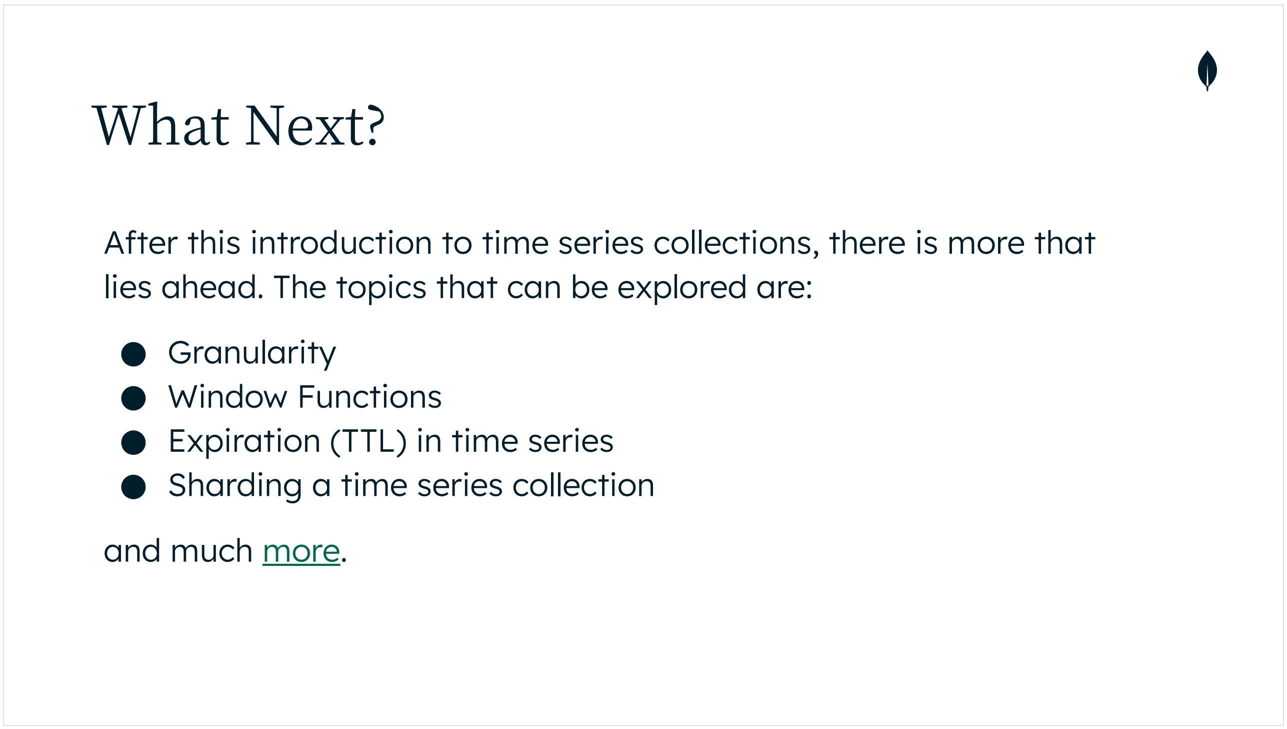
The user just has to work with the above mentioned **abstract layer** while MongoDB takes care of the internal complications.

On the surface, time series collections behave **exactly** like normal collections except that they are more efficient.

We can **insert and query** our data as one normally would.



Under the hood, the creation of a time series collection results in a collection and an automatically created writable non-materialized view which serves as an abstraction layer. This abstraction layer allows you to always work with their data as single documents in their raw form without worry of performance implications as the actual time series collection implements a form of the bucket pattern when persisting data to disk, but these details are something you **no longer need to care about** when designing your schema or reading and writing your data. Users will always be able to work with the abstraction layer and not with a complicated compressed bucketed document.



What Next?

After this introduction to time series collections, there is more that lies ahead. The topics that can be explored are:

- Granularity
- Window Functions
- Expiration (TTL) in time series
- Sharding a time series collection

and much more.

https://www.mongodb.com/developer/products/mongodb/window-functions-and-time-series/

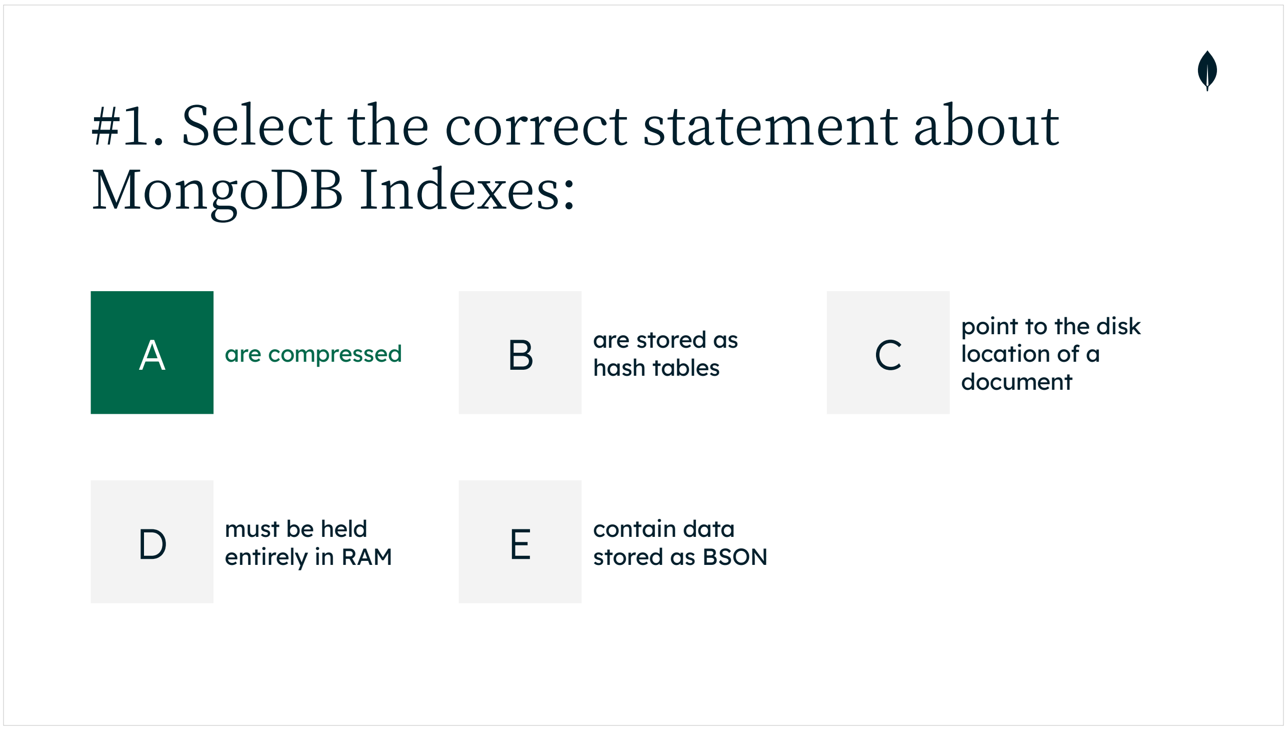
The true value of time series comes from being able to do complex things like an average of the past 5 days.

Quiz Time!

#1. Select the correct statement about MongoDB Indexes:

A	are compressed
B	are stored as hash tables
C	point to the disk location of a document
D	must be held entirely in RAM
E	contain data stored as BSON

Answer in the next slide.



#1. Select the correct statement about MongoDB Indexes:

[✓ A] are compressed

B are stored as hash tables

C point to the disk location of a document

D must be held entirely in RAM

E contain data stored as BSON

Answer: A

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

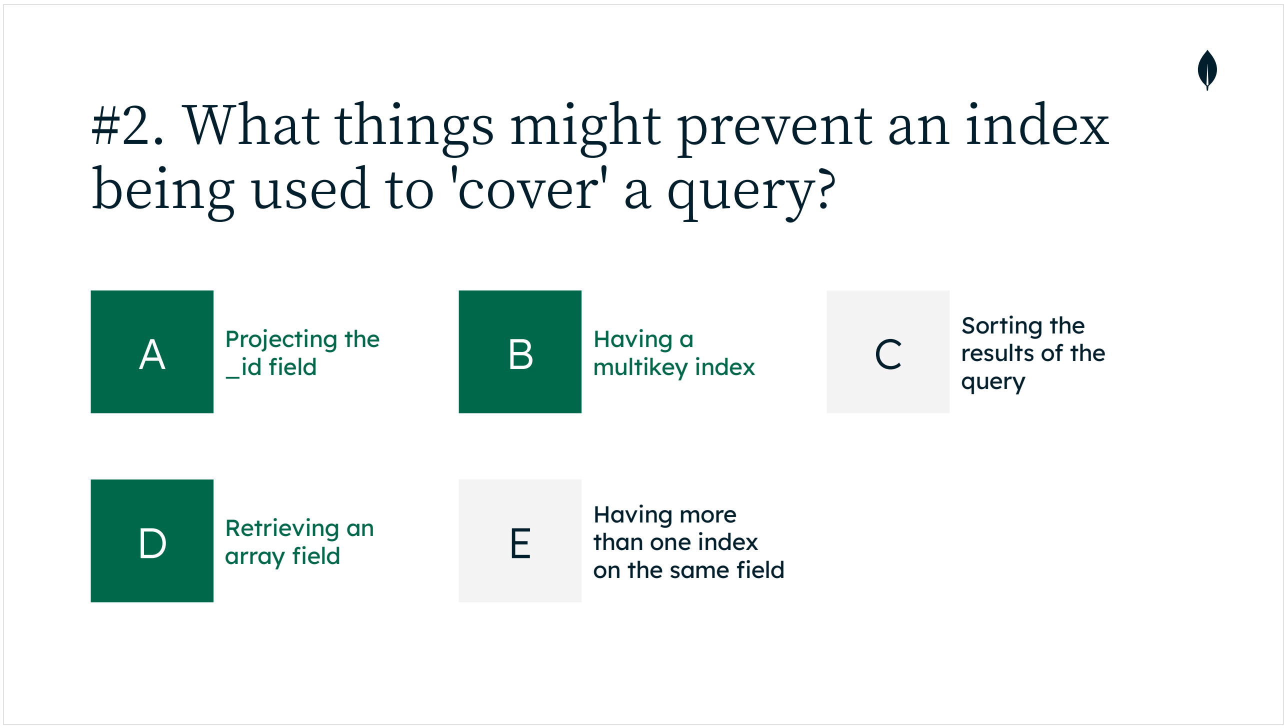
A Projecting the _id field

B Having a multikey index

C Sorting the results of the query

D Retrieving an array field

E Having more than one index on the same field



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

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

Answers A, B, D

#3. Select all true statements about MongoDB.

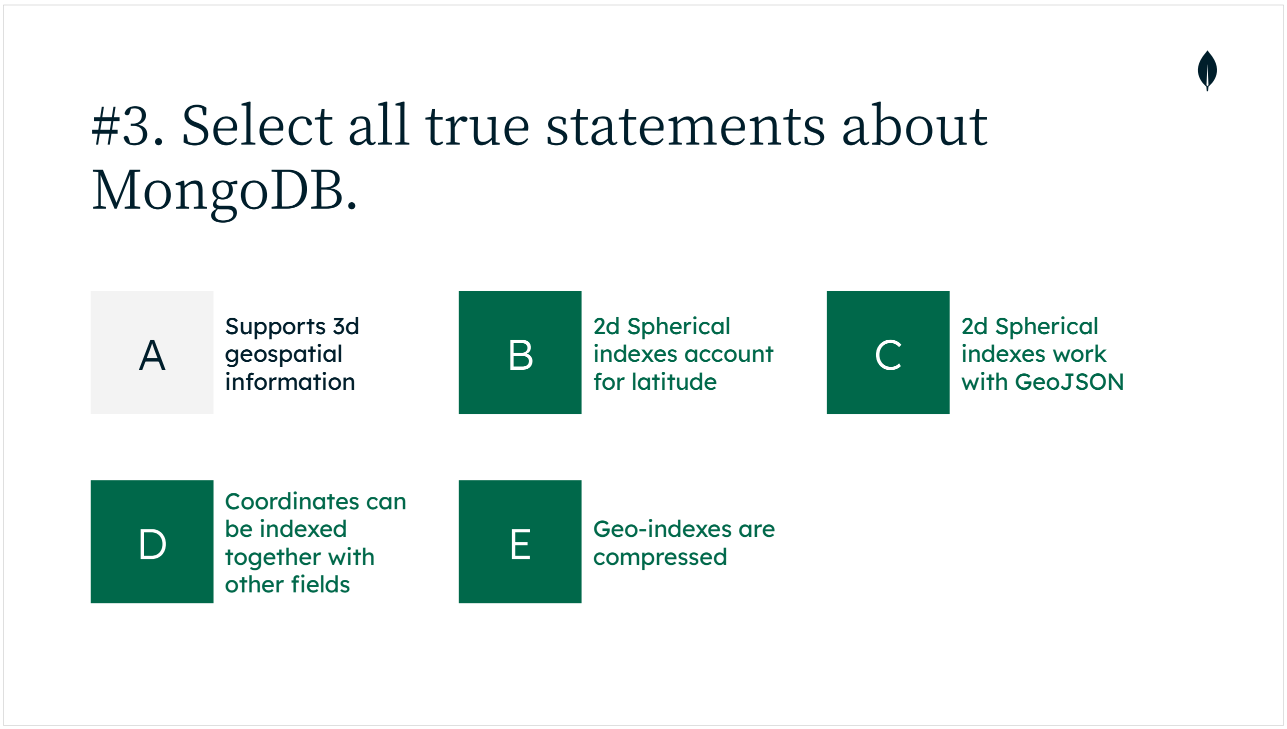
A Supports 3d geospatial information

B 2d Spherical indexes account for latitude

C 2d Spherical indexes work with GeoJSON

D Coordinates can be indexed together with other fields

E Geo-indexes are compressed



#3. Select all true statements about MongoDB.

A — Supports 3d geospatial information

B — 2d Spherical indexes account for latitude

C — 2d Spherical indexes work with GeoJSON

D — Coordinates can be indexed together with other fields

E — Geo-indexes are compressed

Answer: B, C, D & E

#4. Select three actions TTL Indexes can perform:

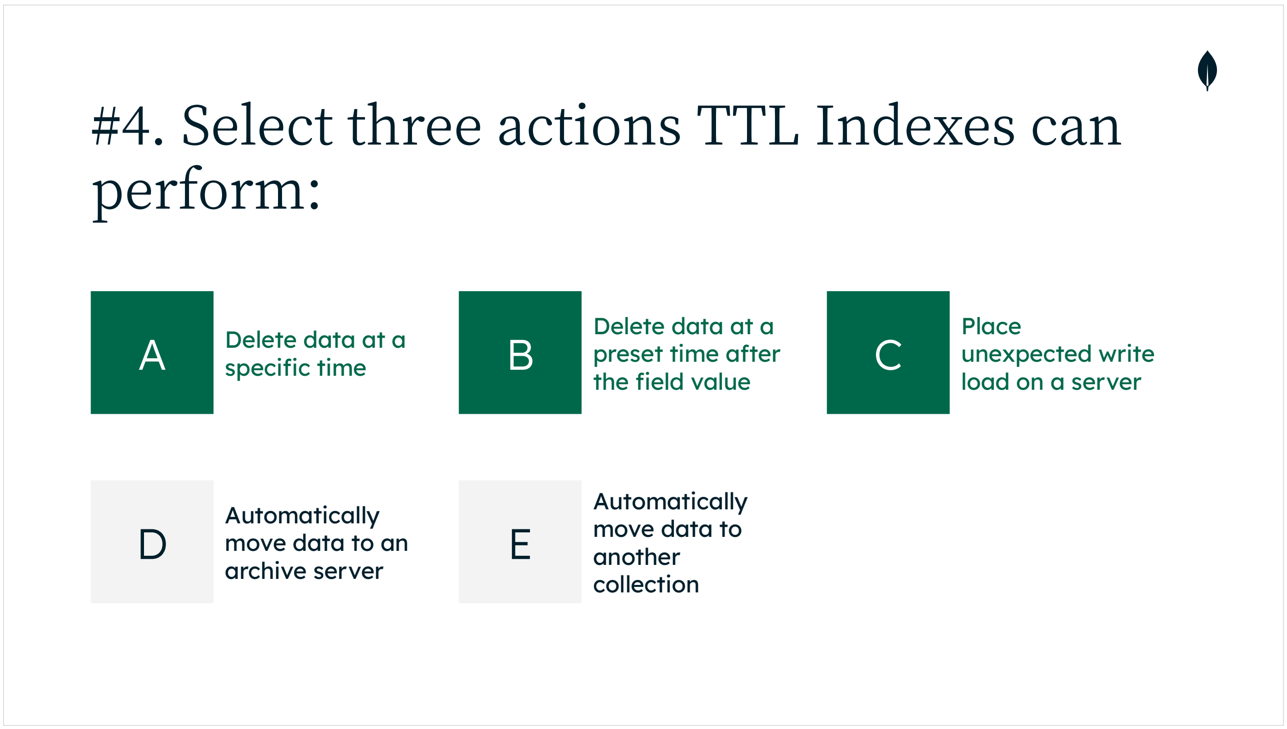
A Delete data at a specific time

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



#4. 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: A, B, C

Recap

Indexes improve efficiency and speed of reads

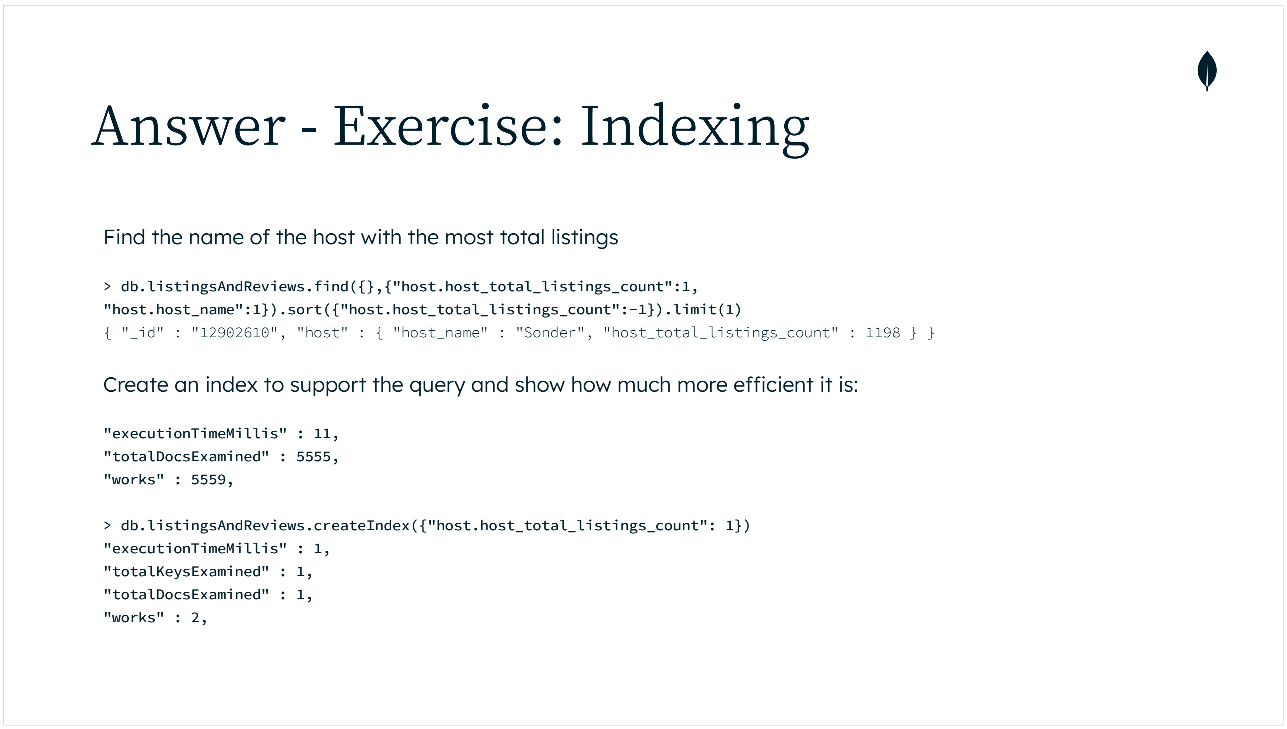
Every query should use an index

Compound indexes are the most used ones in MongoDB

The order in a compound index is very important

MongoDB can explain() how an operation is being indexed

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
```

Exercise Answers

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})
```

Appendix

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.

Even more Index usage tips

Indexes have ~10% overhead on writing 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

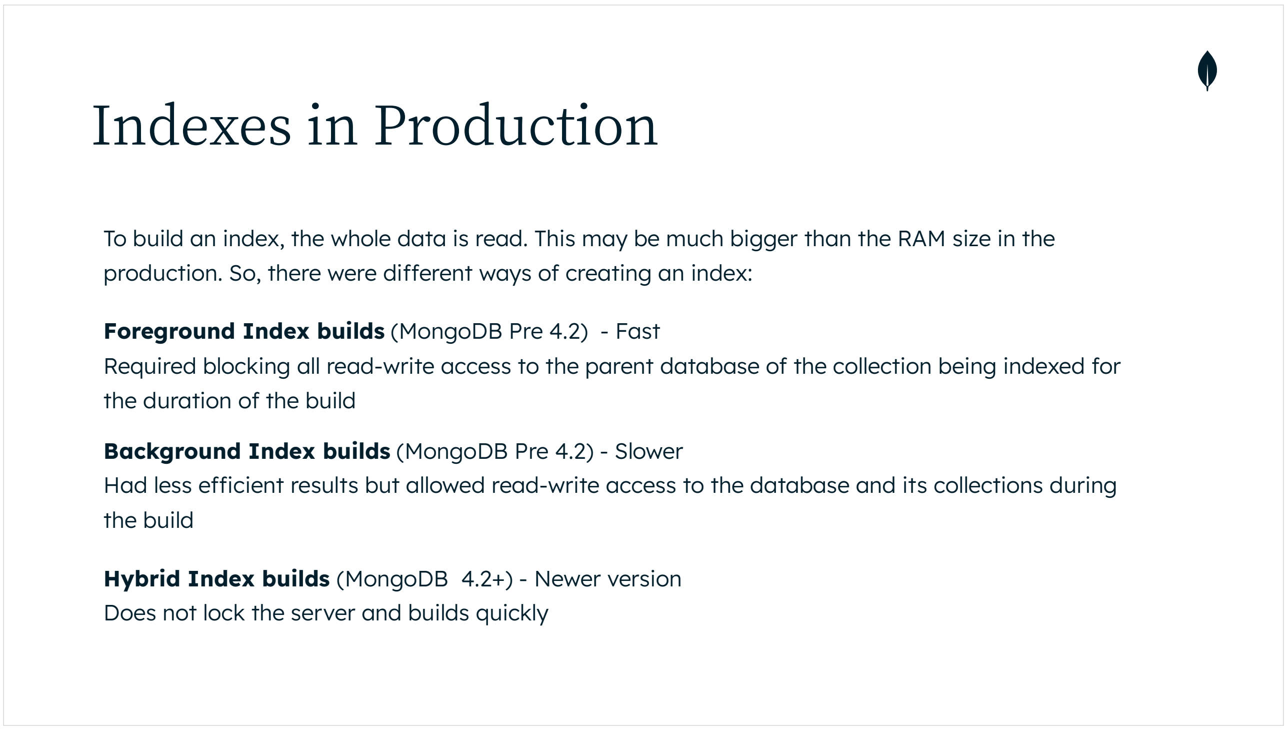
Can use $indexStats() aggregation to see how much each index is used

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.
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.



Indexes in Production

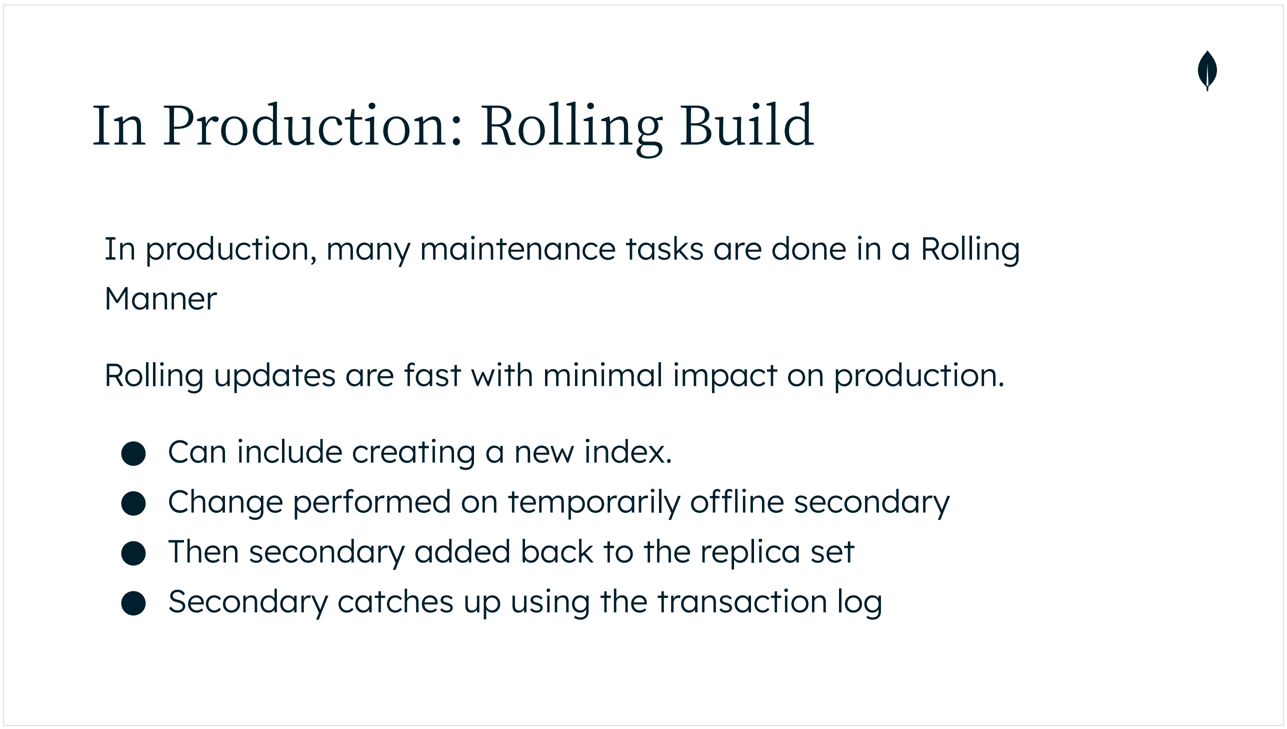
To build an index, the whole data is read. This may be much bigger than the RAM size in the production. So, there were different ways of creating an index:

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

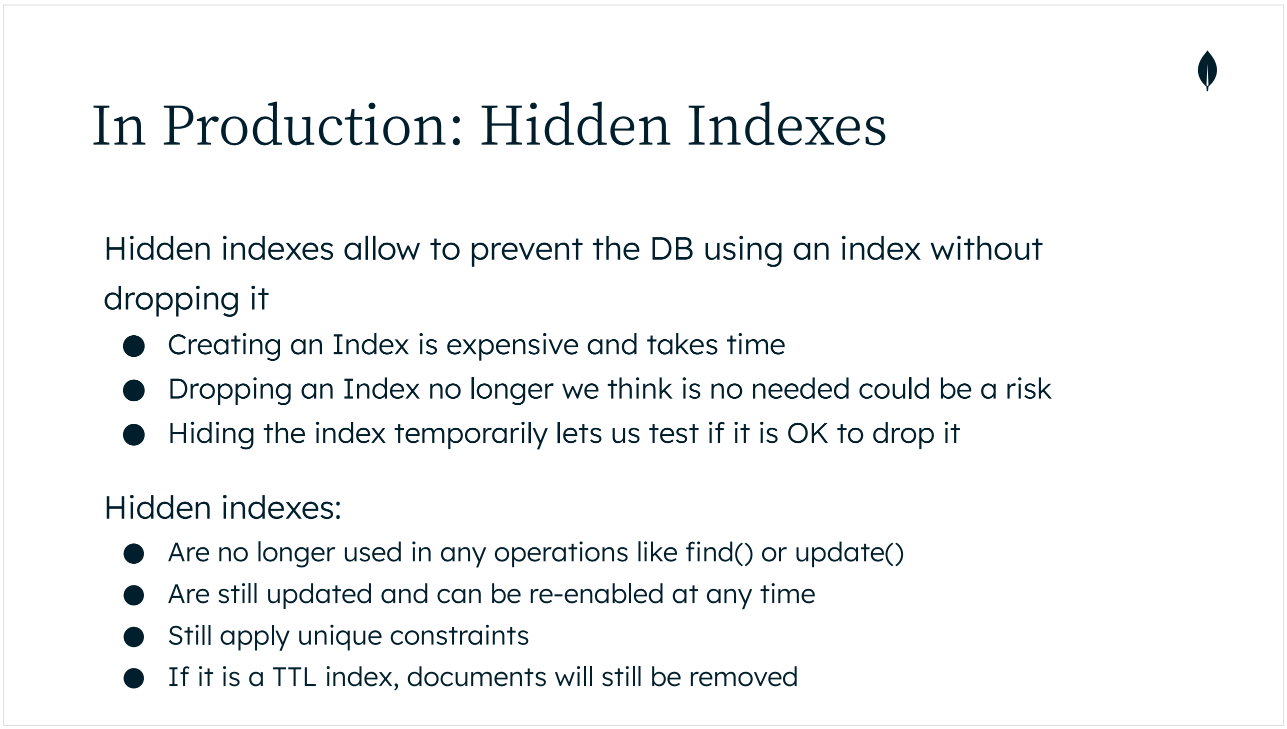
In production, many maintenance tasks are done in a Rolling Manner

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