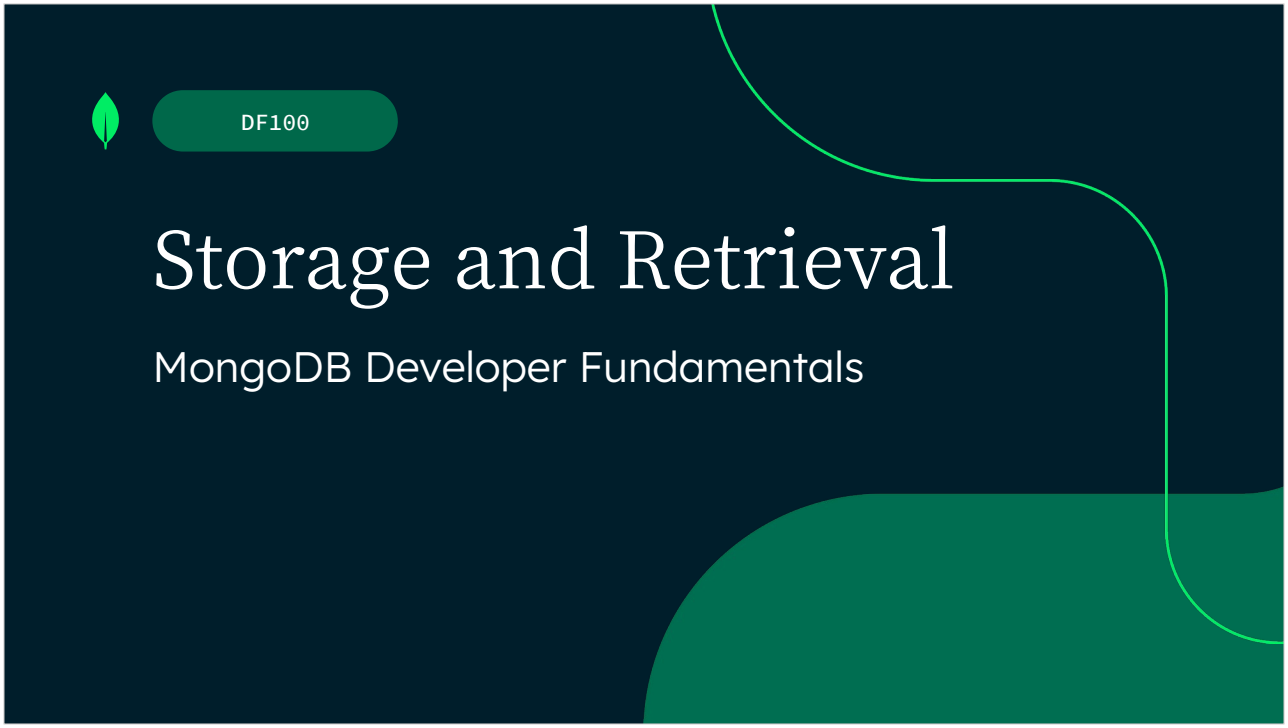


Release: 20230414

Topics we cover

Creating Documents

Cursors

Absolute Changes

Relative Changes

Conditional Changes

Updating Arrays

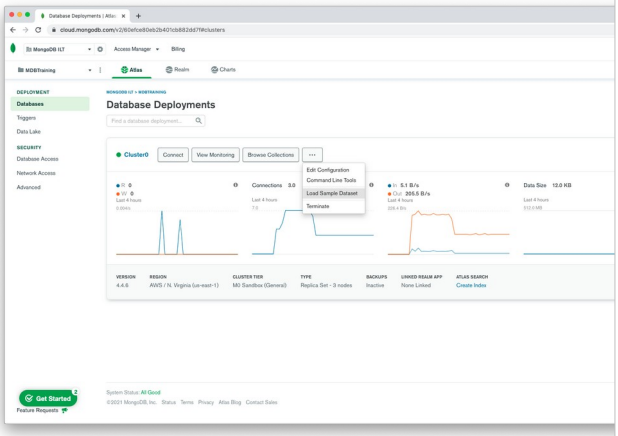
Deleting Documents

Load Sample Data

In your Atlas cluster,
Click the three dots [...]
Select Load Sample Dataset

Click **Browse Collections** to view the databases and collections we loaded.

We will be using the **sample_training** database.



Follow the instructions to load sample data set in Atlas.
Click on the Collections Button to see a list of databases, out of which we would be using the **sample_training** database.

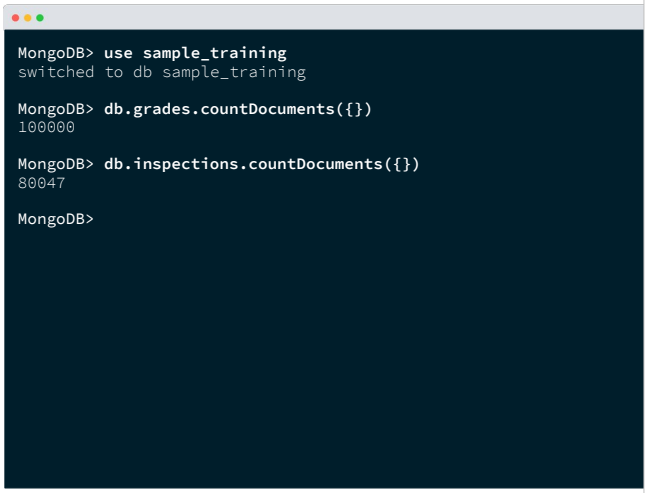
Validate Loaded Data

Connect to the MongoDB Atlas Cluster using mongosh

Verify that data is loaded

Database to use: **sample_training**

Collections to verify: **grades** and **inspections**



Validate the loaded data by checking collection counts for **sample_training.grades** and **sample_training.inspections.**

countDocuments causes the query to return just the number of results found.

Basic Database CRUD Interactions

	Single Document	Multiple Documents
Create	`insertOne(doc)`	`insertMany([doc,doc,doc])`
Read	`findOne(query, projection)`	`find(query, projection)`
Update	`updateOne(query,change)`	`updateMany(query,change)`
Delete	`deleteOne(query)`	`deleteMany(query)`

MongoDB APIs allow us to perform Create Read Update and Delete operations options to perform single or multiple operations.

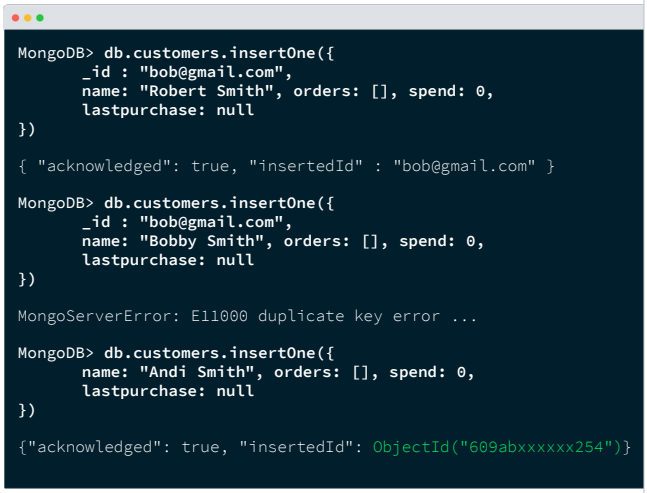
Creating new documents

Creating New Documents - insertOne()

insertOne() adds a document to a collection.

Documents are essentially Objects.

_id field must be unique, it will be added if not supplied.



`insertOne()` adds a document to the collection on which it is called. It is the most basic way to add a new document to a collection.

There are a very few default constraints, the document - which is represented by a language object - **Document, Dictionary, Object must be <16MB**

It must have a **unique value for _id**. If we don't provide one, MongoDB will assign it a GUID of type ObjectId - **a MongoDB GUID type 12 bytes long.**

`{ "acknowledged": true, ... }` means it has succeeded in writing the data to one member of the replica set however we have not specified whether we need it to be on more than one, or even flushed to disk by default.

We can request stronger write guarantees as we will explain later.

Add Multiple Documents - insertMany()

Accepts an array of documents.

Single network call normally.

Reduces network time.

Returns an object with information about each insert.

```
// 1000 Network Calls
MongoDB> let st = ISODate()
for(let d=0;d<1000;d++) {
db.orders.insertOne({ product: "socks", quantity: d})
}
print(`${ISODate()-st} milliseconds`)

9106ms


// 1 Network call, same data
MongoDB> let st = ISODate()
let docs = []
for(let d=0;d<1000;d++) {
      docs.push({ product: "socks", quantity: d})
}
db.orders.insertMany(docs)
print(`${ISODate()-st} milliseconds`)

51ms
```

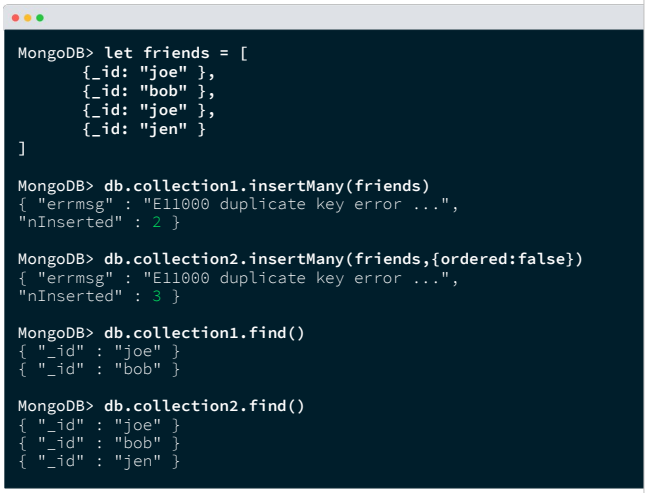
- `insertMany()` can add multiple new documents. Often 1000 at a time.
- This avoids the need for a network round trip per document, which is really slow
- Returns a document showing the success/failure of each and any primary keys assigned
- Limit of 48MB or 100,000 documents data in a single call to the server, but a larger batch is broken up behind the scenes by the driver
- There is a way to bundle Insert, Update and Delete operations into a single network call too called BulkWrite.

Order of operations in insertMany()

insertMany() can be ordered or unordered.

Ordered (default) stops on first error.

Unordered reports errors but continues; can be reordered by the server to make the operation faster.



- If we opt for strict ordering then:
  - It must stop on first error
  - No reordering or parallelism can be done so slower in sharded cluster.

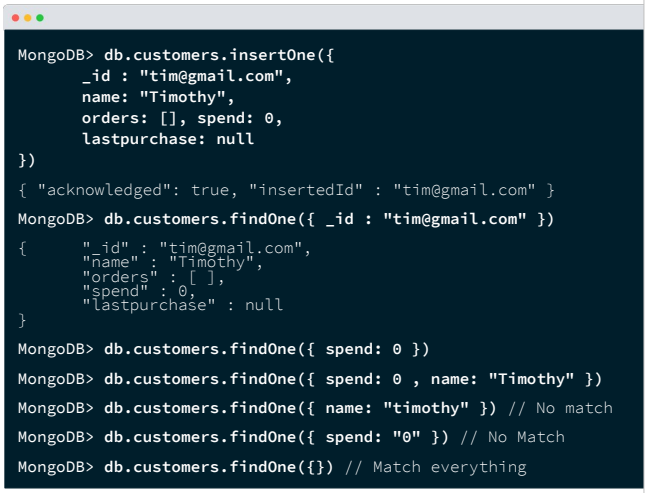
Reading documents

Find and Retrieve documents

findOne() retrieves a single document.

Accepts a **document** as a filter to “query-by-example.”

Empty object (or no object) matches everything.



We can retrieve a document using `findOne(). findOne()` takes an Object as an argument

We return the first document we find where all the members match. If there are multiple matches there is no way to predict which is 'first' in this case.

Here we add a record for customer Timothy using `insertOne()`

Then we query by the `_id` field - which has the user’s email and we find the record - this returns an object - and mongosh prints what is returned.

We can also query by any other field - although only `_id` has an index by default so the others here are less efficient for now.

We can supply multiple fields, and if they all match we find the record - Someone called Timothy who has spent 0 dollars.

Note that the order of the fields in the query does not matter here - we can think of the comma as just meaning **AND**

`db.customers.findOne({ spend: "0" })` fails - because it's looking for the String "0" not the number 0 so doesn't match.

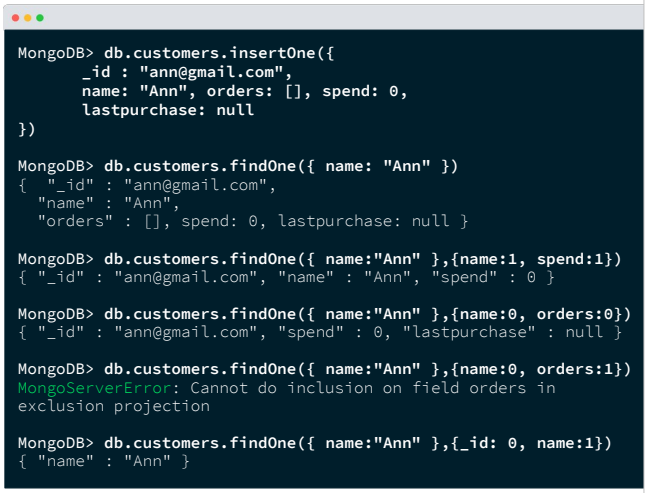
An Empty object matches everything. However, due to the inherent nature of findOne() it would return us only one document.

Projection: choosing the fields to return

Find operations can include a **projection** parameter.

Projections only return a subset of each document.

Projections include/exclude a set of fields.



```
MongoDB> db.customers.insertOne({
       _id : "ann@gmail.com",
       name: "Ann", orders: [], spend: 0,
       lastpurchase: null
})

MongoDB> db.customers.findOne({ name: "Ann" })
{  "_id" : "ann@gmail.com",
  "name" : "Ann",
  "orders" : [], spend: 0, lastpurchase: null }

MongoDB> db.customers.findOne({ name:"Ann" },{name:1, spend:1})
{ "_id" : "ann@gmail.com", "name" : "Ann", "spend" : 0 }

MongoDB> db.customers.findOne({ name:"Ann" },{name:0, orders:0})
{ "_id" : "ann@gmail.com", "spend" : 0, "lastpurchase" : null }

MongoDB> db.customers.findOne({ name:"Ann" },{name:0, orders:1})
MongoServerError: Cannot do inclusion on field orders in
exclusion projection

MongoDB> db.customers.findOne({ name:"Ann" },{_id: 0, name:1})
{ "name" : "Ann" }
```

We can select the fields to return by providing an object with those fields and a value of 1 for each.

Documents can be large; with the help of projection we can have MongoDB return a subset of the fields.

_id is always returned by default.

We can instead choose what field NOT to return by providing an object with fields set to 0.

We cannot mix and match 0 and 1 - as what should it do with any other fields?

There is an exception where we can use **_id: 0** it to remove **_id** from the projection and project only the fields that are required `{ _id:0, name : 1 }`

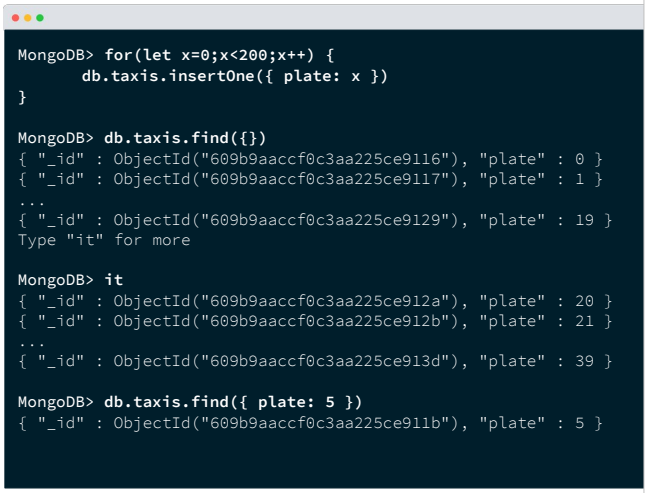
There are some more advanced projection options, including projecting parts of an array and projecting computed fields using aggregation but those are not covered here.

Fetch multiple documents using find()

find() returns a cursor object rather than a single document

We fetch documents from the cursor to get all matches

mongosh fetches and displays 20 documents from the cursor object.



Find returns a cursor object, by default the shell then tries to print that out.

The cusros object prints out by displaying its next 20 documents and setting the value of a variable called it to itself.

If we type `it` - then it tries to print the cursor again - and display the next 20 objects.

As a programmer - cursors won't do anything until we look at them.

We can add `.pretty()` to a cursor object to make the shell display larger documents with newlines and indentation.

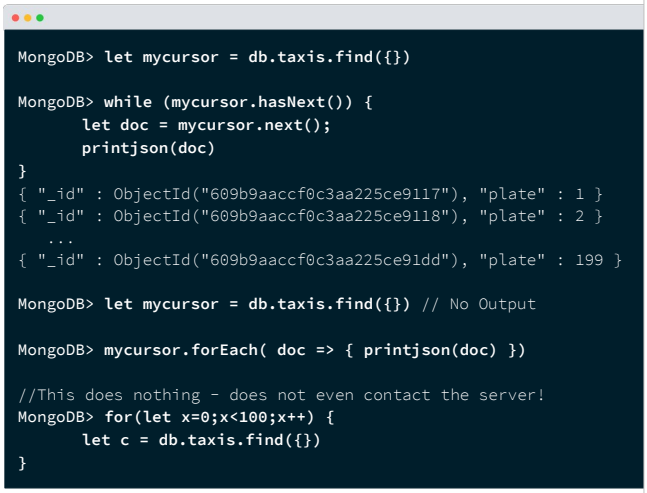
Cursors

Using Cursors

Here, we store the result of find to a variable.

We then manually iterate over the cursor.

The query is not actually run until we fetch results from the cursor.



```
MongoDB> let mycursor = db.taxis.find({})

MongoDB> while (mycursor.hasNext()) {
       let doc = mycursor.next();
       printjson(doc)
}
{ "_id" : ObjectId("609b9aaccf0c3aa225ce9117"), "plate" : 1 }
{ "_id" : ObjectId("609b9aaccf0c3aa225ce9118"), "plate" : 2 }
   ...
{ "_id" : ObjectId("609b9aaccf0c3aa225ce91dd"), "plate" : 199 }

MongoDB> let mycursor = db.taxis.find({}) // No Output

MongoDB> mycursor.forEach( doc => { printjson(doc) })

//This does nothing - does not even contact the server!
MongoDB> for(let x=0;x<100;x++) {
       let c = db.taxis.find({})
}
```

`mycursor` is a cursor object, it knows the database, collection and query we want to run.

Until we do something with it it has not run the query - it has not even contacted the server.

It has methods - importantly , in mongosh `hasNext()` and `next()` to check for more values and fetch them.

We can iterate over a cursor in various ways depending on our programming language.

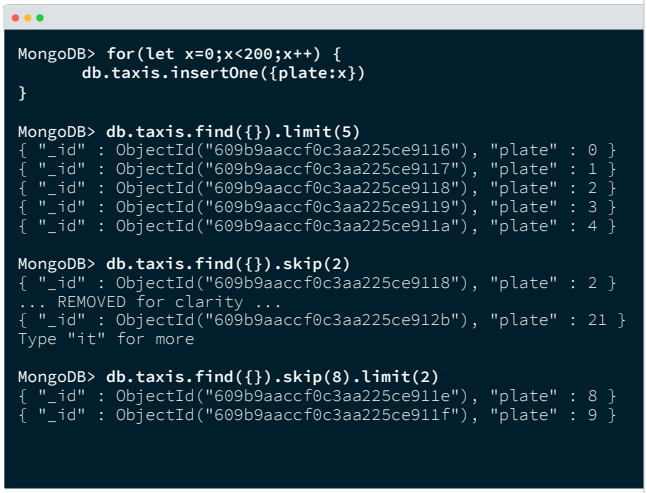
If we don't fetch information from a cursor - it never executes the find - this might not be expected when doing simple performance tests like the one below.

To pull the results from a cursor in a shell for testing speed we can use `db.collection.find(query).itcount()`

Cursor modifiers

Cursors can include additional instructions like **limit**, **skip**, etc.

Skip and limit return us cursors.



```
MongoDB> for(let x=0;x<200;x++) {
       db.taxis.insertOne({plate:x})
}

MongoDB> db.taxis.find({}).limit(5)
{ "_id" : ObjectId("609b9aaccf0c3aa225ce9116"), "plate" : 0 }
{ "_id" : ObjectId("609b9aaccf0c3aa225ce9117"), "plate" : 1 }
{ "_id" : ObjectId("609b9aaccf0c3aa225ce9118"), "plate" : 2 }
{ "_id" : ObjectId("609b9aaccf0c3aa225ce9119"), "plate" : 3 }
{ "_id" : ObjectId("609b9aaccf0c3aa225ce911a"), "plate" : 4 }

MongoDB> db.taxis.find({}).skip(2)
{ "_id" : ObjectId("609b9aaccf0c3aa225ce9118"), "plate" : 2 }
... REMOVED for clarity ...
{ "_id" : ObjectId("609b9aaccf0c3aa225ce912b"), "plate" : 21 }
Type "it" for more

MongoDB> db.taxis.find({}).skip(8).limit(2)
{ "_id" : ObjectId("609b9aaccf0c3aa225ce911e"), "plate" : 8 }
{ "_id" : ObjectId("609b9aaccf0c3aa225ce911f"), "plate" : 9 }
```

We can add a limit instruction to the cursor to stop the query when it finds enough results.

We can add a skip instruction to the cursor to tell it to ignore the first N results.

The Skip is always performed before the limit when computing the answer.

This can be used for simple paging of results - although it's not the optimal way of doing so.

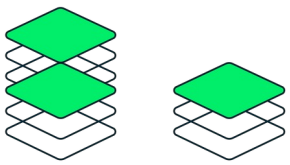
Skip has a cost on the server - skipping a large number of documents is not advisable.

Cursors work in batches

Cursors fetch results from the server in batches.

The default batch size in the shell is 101 documents during the initial call to find() with a limit of 16MB.

If we fetch more than the first 100 document from a cursor it fetches in 16MB batches in the shell or up to 48MB in some drivers.



Rather than make a call to the server every time we get the next document from a cursor, the server fetches the result in batches and stores them at the client or shell end until we want them.

Fetching documents one by one would be slow.

Fetching all documents at once would use too much client RAM.

We can change the batch size on the cursor if we need to but it's still limited to 16M.

Fetching additional data from a cursor uses a function called getmore() behind the scenes, it fetches 16MB at a time.

Exercise

Add four documents to a collection called diaries using the commands shown here.

Write a find() operation to output only diary entries from **dug**.

Modify it to output the line below using **skip**, **limit** and a **projection**.

```
{ name: "dug", txt: "saw a squirrel" }
```

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

MongoDB> db.diaries.insertMany([
      {
            name: "dug", day: ISODate("2014-11-04"),
            txt: "went for a walk"
      },
      {
            name: "dug", day: ISODate("2014-11-06"),
            txt: "saw a squirrel"
      },
      {
            name: "ray", day: ISODate("2014-11-06"),
            txt: "met dug in the park"
      },
      {
            name: "dug", day: ISODate("2014-11-09"),
            txt: "got a treat"
      }
])
```

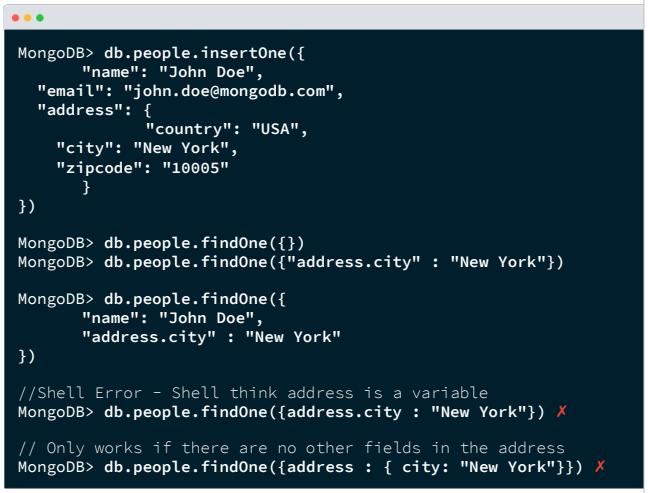
Answers at the end

Querying values in nested documents

Fields can contain documents

Use dot notation to specify a field in a nested document: **“address.city”**

In mongosh put the field names in quotes.



```
MongoDB> db.people.insertOne({
       "name": "John Doe",
   "email": "john.doe@mongodb.com",
   "address": {
              "country": "USA",
     "city": "New York",
     "zipcode": "10005"
       }
})

MongoDB> db.people.findOne({})
MongoDB> db.people.findOne({"address.city" : "New York"})

MongoDB> db.people.findOne({
       "name": "John Doe",
       "address.city" : "New York"
})

//Shell Error - Shell think address is a variable
MongoDB> db.people.findOne({address.city : "New York"}) ✗

// Only works if there are no other fields in the address
MongoDB> db.people.findOne({address : { city: "New York"}}) ✗
```

In MongoDB we can have fields which are documents in addition to the basic scalar types like string or integer.

If we want to query a field that’s a member of a field - we need to use a dot notation, in quotes when referring to the name.

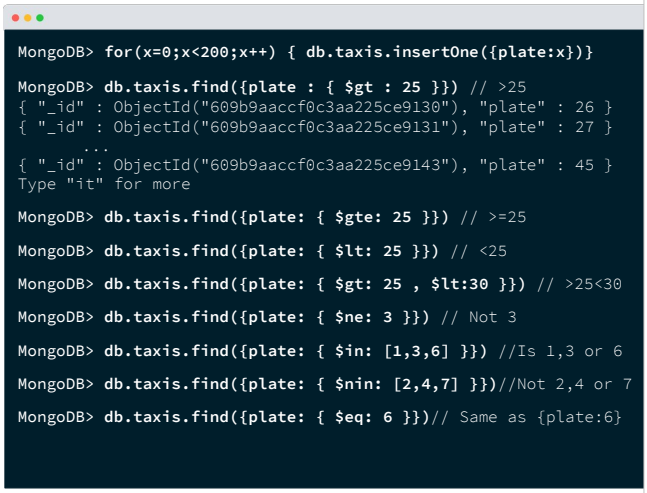
If we don't have quotes in the shell then JavaScript thinks we are dereferencing a variable when we do address.city

In the bottom example - we are are comparing Address as a whole to an object - which will only work if the object is an exact match.

Query by ranges of values

There are operators to compare relative values like greater or less than.

Can also check an explicit set of values using $in - true if the value is in the list.



MongoDB has relative comparison operators like greater than and less than.

To use these we compare the field to an object with a dollar operator instead of a value so { a: { $gt: 5}} not { a : 5 }

Actually `{ a: 5 }` is shorthand for `{ a : { $eq : 5} }` the equality operator.

If we do `{ a: { $gt:5}, a: {$lt: 8} }` ✗ in our programming language or the shell , this is actually just entering `{ a:{$lt:8} }`

Correct version is `{ a: { $gt:5, $lt:8 }}`

Boolean Logic Operators

Logic operators in queries: AND, OR, NOR, and NOT

Take an array as value and can have more than two clauses

These are normally used with complex clauses

```
MongoDB> db.pets.insertMany([
       { species: "cat", color: "brown"},
       { species: "cat", color: "black"},
       { species: "dog", color: "black"},
       { species: "dog", color: "brown"},
])

MongoDB> db.pets.find({
       $or: [
              {species:"cat",color:"black"},
              {species:"dog",color:"brown"}
       ]
})
{ "_id" : ObjectId("..."), "species" : "cat", "color" : "black" }
{ "_id" : ObjectId("..."), "species" : "dog", "color" : "brown" }

//Black pets that are not less than cats (alphabetically)
MongoDB> db.pets.find({
       species: {
              $not: {$lte: "cat" }
       },
       color: "black"
})
```

We can use `$and, $or, $nor` and `$not` to combine more complex query clauses - `$or` is the one most commonly used.

Exercise - Range and Logic

1. In the MongoDB shell, change to using the database **sample_training**
2. In the **grades** collection, see how many documents have **student_id** less than or equal to 65
3. In the **inspections** collection, see how many documents have **result** as "Pass" or "Fail" (Write this in two different ways)

To change database type `use sample_training` in the shell
Answer at the end

Querying Values in Arrays

When the query field is an Array, a document is returned if query:

- Matches any array member
- Exactly matches the whole array, including the order

```
MongoDB> db.fun.insertOne({
       "name": "John",
       hobbies: ["cars","robots","gardens"]
})
{"acknowledged":true }

//Find by ANY member of the array
MongoDB> db.fun.find({ hobbies: "gardens" })
{"_id":ObjectId("..."),"name":"John","hobbies":
["cars","robots","gardens"]}

//Find by matching the array itself
MongoDB> db.fun.find({hobbies: ["cars","robots","gardens"]})
{"_id":ObjectId("..."),"name":"John","hobbies":
["cars","robots","gardens"]}

//Not found - order doesn't match
MongoDB> db.fun.find({ hobbies: ["robots","cars","gardens"] })

//Not found - missing element
MongoDB> db.fun.find({ hobbies: ["cars","robots"] })
```

When querying an array with standard find syntax, you either match one element of the array or the whole array (as per slide)

This is a "contains" query

There are other array query operators, including querying computed values.

All the operations we have seen like $gt and $in work in the same way against arrays.

Array specific query operators

MongoDB has operators designed specifically for querying against arrays.

These are as follows:

- $all
- $size
- $elemMatch

Why is there no $any operator?

```
MongoDB> db.fun.insertOne({
      "name": "John",
      hobbies: ["cars","robots","gardens"]
})
{"acknowledged":true }

MongoDB> db.fun.find({
      hobbies: { $all: ["robots","cars"] }
})
{ "_id" : ObjectId("..."), "name" : "John", "hobbies" : [
"cars", "robots", "gardens" ] }

MongoDB> db.fun.find({
      hobbies: { $all: ["robots", "cars", "bikes"] }
}) //No result as bikes is not in the array

MongoDB> db.fun.find({ hobbies : { $size : 3}})
{ "_id" : ObjectId("..."), "name" : "John", "hobbies" : [
"cars", "robots", "gardens" ] }

MongoDB> db.fun.find({ hobbies : { $size : 4 } })
```

`$all` takes a list of values and matches where the array contains all of those values - there may be additional values in the array and order doesn't matter.

`$size` matches if the array length is exactly the size specified. You cannot use it with `$gt` or `$lt`.

We will cover `$elemMatch` in the next slide.

Why do you think there is no `$any` operator to find where the array has any of a list of values?

A surprising array query

If we have a query like this:

```
{ age: { $gt : 18 , $lt: 30 } }
```

Expecting to match **age** between 18 and 30 - But what if **age** is an array?

```
{ age: [ 40, 10, 5 ]  }
```

Would it match this?

If so, why?

Using $elemMatch

To limit it to a single element use a query like this:

```
{ age : {$elemMatch: { $gt : 18 , $lt: 30 } }}
```

Read $elemMatch as “has an element that matches the query”

In this case, the query is:

```
{ $gt : 18 , $lt: 30 }
```

Not using $elemMatch when required is a common source of errors in MongoDB queries

When we match against an array, we check that each requirement matches at least one element of the array - but it doesnt need to be the same element.
For that we have to use $elemMatch

Exercise: $elemMatch

In **sample_restaurants.restaurants**, there are hygiene ratings and dates of inspection details.

Executing this query to find restaurants that have had a hygiene rating of C after 2013 finds 2,675 restaurants:

```
db.restaurants.find({
  "grades.grade":"C",
   "grades.date":{$gt:ISODate("2013-12-31")}
})
```

This is incorrect as finds one that hasn't had a C since 2014. What would be the correct query?

Try this query to get the shape of a restaurant with some relevant data.

```
MongoDB> db.restaurants.findOne()
{
    "_id" : ObjectId("5eb3d668b31de5d588f4292a"),
    "address" : {
        "building" : "2780",
        "coord" : [
            -73.98241999999999,
            40.579505
        ],
        "street" : "Stillwell Avenue",
        "zipcode" : "11224"
    },
    "borough" : "Brooklyn",
    "cuisine" : "American",
    "grades" : [
        {
            "date" : ISODate("2014-06-10T00:00:00Z"),
            "grade" : "A",
            "score" : 5
        },
        {
            "date" : ISODate("2013-06-05T00:00:00Z"),
            "grade" : "A",
            "score" : 7
        },
        {
            "date" : ISODate("2012-04-13T00:00:00Z"),
            "grade" : "A",
            "score" : 12
```

Sorting Results

Use **sort()** cursor modifier to retrieve results in a specific order

Specify an object listing fields in the order to sort and sort direction

```
MongoDB> let rnd = (x)=>Math.floor(Math.random()*x)

MongoDB>
for(let x=0;x<100;x++) { db.scores.insertOne({ride:rnd(40),swim:
rnd(40),run:rnd(40)})}

//Unsorted
MongoDB> db.scores.find({},{_id:0})
{ "ride" : 5, "swim" : 11, "run" : 11 }
{ "ride" : 0, "swim" : 17, "run" : 12 }
{ "ride" : 17, "swim" : 2, "run" : 2 }

//Sorted by ride increasing
MongoDB> db.scores.find({},{_id:0}).sort({ride:1})
{ "ride" : 0, "swim" : 38, "run" : 10 }
{ "ride" : 1, "swim" : 37, "run" : 37 }
{ "ride" : 1, "swim" : 30, "run" : 20 }

//Sorted by swim increasing then ride decreasing
MongoDB> db.scores.find({},{_id:0}).sort({swim:1, ride:-1})
{ "ride" : 31, "swim" : 0, "run" : 14 }
{ "ride" : 11, "swim" : 0, "run" : 14 }
{ "ride" : 30, "swim" : 1, "run" : 34 }
{ "ride" : 21, "swim" : 1, "run" : 3 }
```

With Skip and Limit sorting can be very important so we skip to limit to what we expect.

We cannot assume anything about the order of unsorted results.

Sorting results without an index is very inefficient - we cover this when talking about indexes later.

Exercise: Combined Query

Use the **sample_training.companies** collection to find the largest company (by number of employees) that has fewer than 200 employees.

Write a query that prints out only the company name and number of employees.

1. Use find() with a query and a projection
2. Apply some of the cursor operators: sort(), skip() and limit()

Answers at the end

Expressive Queries

$expr can query with Aggregations

$expr can match ANY computable function in the data.

$expr only uses indexes for equality match of a constant value before MongoDB 5.0.

```
MongoDB> use sample_mflix
switched to db sample_mflix

//Movies where rotten tomatoes rates it higher than imdb
MongoDB> db.movies.find({
   $expr: { $gt: [ "$tomatoes.viewer.rating" ,"$imdb.rating" ] }

})

MongoDB> use sample_training
switched to db sample_training

//Grades where average score < 50
MongoDB> db.grades.find({
    $expr: { $lt: [
        { $avg: "$scores.score" },
        50
    ]}
})
```

- Aggregation will be covered later but the example lets you compare fields or even computed fields (e.g. where array has any value that is greater than 2x the average in the array)
- This allows us to compare values inside a document to each other
- Or to calculate something like "find where width*height > 50"
- So needs to be used with care to ensure it doesn't slow the system.
- $expr is available from the version MongoDB 3.6
- $expr only uses indexes for Exact matches before MongoDB 5.0. it cannot use them for range or other queries. After 5.0 it can use them for $gt,$lt,$gte and $lte
- $expr doesn't support multi-key indexes. So needs to be used with care to ensure it doesn't slow the system.

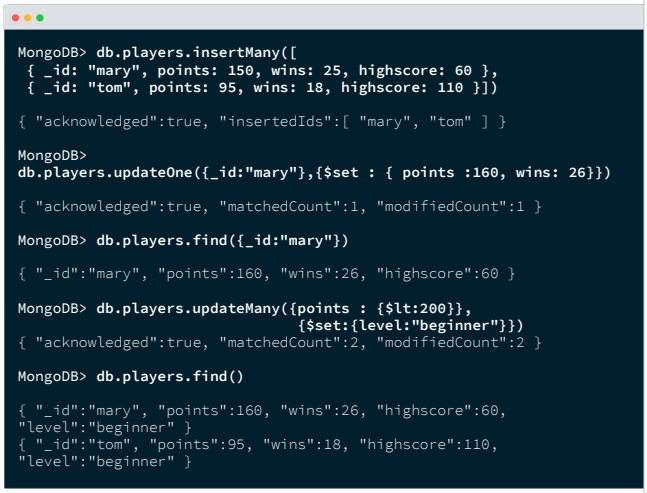
Updating documents

Updating Documents

Modify documents using updateOne or updateMany

updateOne(query, change) changes only the first matching document

updateMany(query,change) changes all matching Documents.



We are demonstrating the basic principle here - find one or more documents and set the value of fields in them.
updateMany is not atomic - it's possible it may stop part way through, if a server fails or if it hits an error condition and then only some records are changed.
updateMany is many updateOne operations - but unlike insertMany it's actually a single request to the server as we are asking for one change that changes many records the same way.

Describing a Mutation

updateOne(query, **mutation**)

Mutation is an object describing the changes to make to each record.

Values can be explicitly set or changed relative to the current value or external values.

The format is:

```
{ operator1 : { field1: value, field2: value},
 operator2 : { field3: value, field4: value }  }
```

We have seen a simple example of a mutation where we used the $set operator to explicitly set the value of a single field - MongoDB update operators can do far more than that though.

The $set operator

$set - sets the value of a field to an explicit absolute value

Use dot notation to set a field in an embedded document

Setting a field to an object replaces the existing value entirely

```
{ $set  :
  {
    length: 10,
    width: 10,
    shape: "square",
    coordinates: [3,4]
  }
}
```

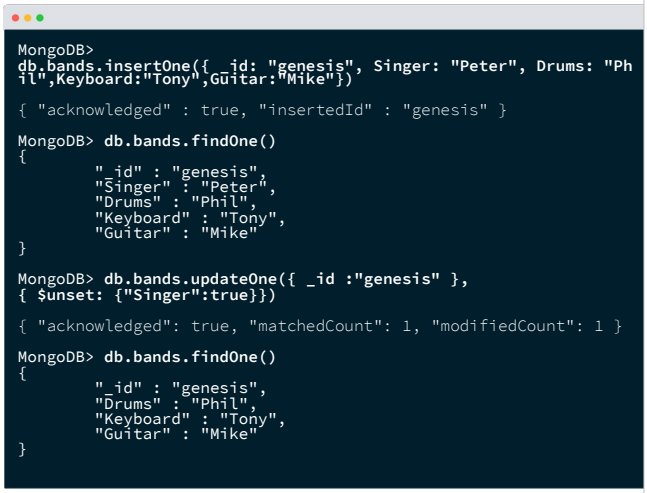
```
{ $set  :
  {
   "schoolname" : "Valley HS",
   staff: { principal: "jones" },
   "address.zip" : 90210
  }
}
```

The $unset operator

Remove a field from a document

Logically equal to set the field to null but takes no storage

$unset takes an object with the fields to remove and a value of 1 or true



Here we can use unset to remove the field Singer - some might say we should $set to set Singer to "Phil" but that is debatable.

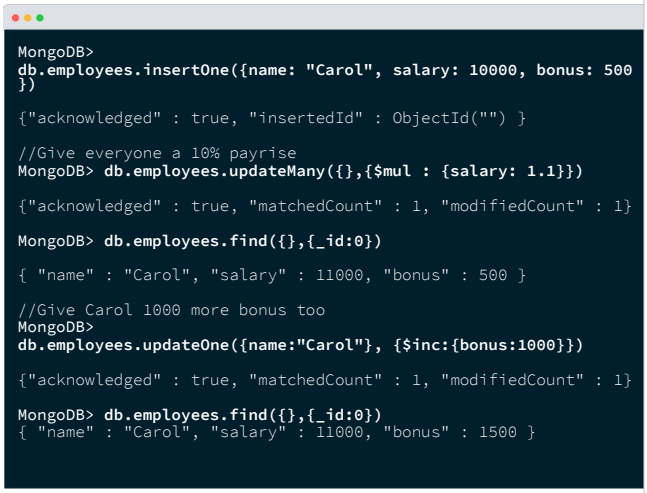
Note: The value of "Singer" in the `$unset: {"Singer":true}` expression doesn't actually matter as it would remove the field anyways.

Relative numeric updates $inc and $mul

$inc and $mul modify numeric value relative to its current value.

$inc changes it by adding a value to it - the value may be negative.

$mul changes it by multiplying it by a value, which may be less than 1



We can pass a numeric mutation operation to the server which will change a value relative to its current value.

This is important - if we were to read the current value to the client then use $set the increased value we have a risk of a race condition.

What if between us reading it and writing it someone else changes it, our change is not relative to the value at the time the edit happens.

Using $inc therefore ensures that the change is relative to the current value. This is an example of the safe, atomic updates that are required to work under load.

In an RDBMS we would pass SET V = V+1 but that would be calculated at the server side - we have an explicit operator for this.

As we will see later we can also so it with an expression like the SQL command though.

Relative value operators $max and $min

$max and **$min** can modify a field depending on its current value.

Only update the field if the value given is larger (or smaller) than the current value.

```
MongoDB>
db.gamescores.insertOne({name: "pacman", highscore: 10000 })

{"acknowledged" : true,"insertedId" : ObjectId("")}

//This finds the record but does not change it as 9000 < 10000
MongoDB>
db.gamescores.updateOne({name:"pacman"},{$max: { "highscore": 90
00}})

{"acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0}

//This finds and changes highscore as 12000 > 10000
MongoDB>
db.gamescores.updateOne({name:"pacman"},{$max: { "highscore": 12
000}})

{"acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1}

MongoDB> db.gamescores.find({})
{ "_id" : ObjectId("609bf0f8cf0c3aa225ce9314"), "name" :
"pacman", "highscore" : 12000 }
```

$min and $max only change the value if changing it would make it smaller ($min) or larger ($max) respectively - so they allow us to easily keep track of the largest value we have seen.In this example a high score.

We could have included the highscore in the query - find only if highscore is less than the score we have, but it may be we ant to make other changes to this record - record this as the 'latest score' but only change 'highscore' when appropriate.

Exercise: Updates

Using the **sample_training.inspections** collection complete the following exercises:

Exercise 1: Pass Inspections
In the collection, let's do a little data cleaning: Replace the "Completed" inspection result to use only "No Violation Issued" for those inspections. Update all the cases accordingly.

Exercise 2: Set fine value
For all inspections that fail, with result "Fail", by setting a new fine field with value 100.

Exercise 3: Increase fines in ROSEDALE
Update all failed inspections done in the city of "ROSEDALE", by raising the "fine" value by 150.

Basic Array updates - $push and $pop

Modify an array that is in a document without having to read and replace the entire array

This is important to prevent overwriting other users changes

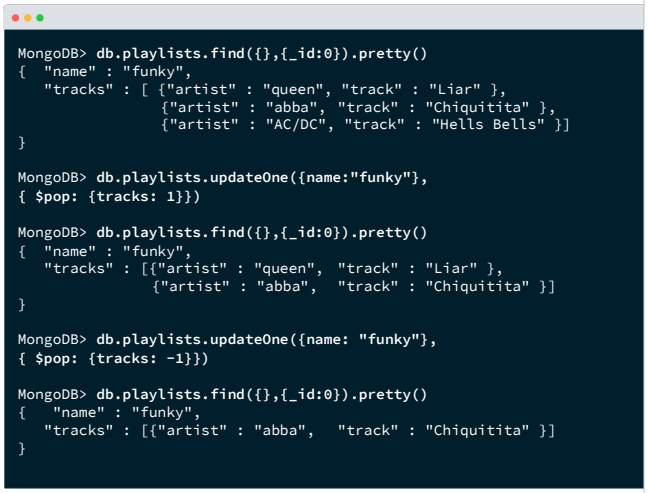
Add and remove items in a number of ways: $push and $pop

```
MongoDB> db.playlists.insertOne(
{name: "funky",
 tracks : [
  { artist:"queen",track:"Liar"},
  {artist:"abba",track:"Chiquitita"},
 ]})

{"acknowledged" : true,"insertedId" : ObjectId("")}

MongoDB> db.playlists.updateOne({name:"funky"},
{ $push : {tracks : { artist: "AC/DC", track: "Hells Bells"} }})

{"acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1}

MongoDB> db.playlists.find({}).pretty()
{   "_id" : ObjectId(""),
   "name" : "funky",
   "tracks" : [
       {"artist" : "queen",       "track" : "Liar" },
       {"artist" : "abba", "track" : "Chiquitita" },
       {"artist" : "AC/DC",       "track" : "Hells Bells" } ]
}
```

$push adds a value to the end of the array - although there is a position options if we wish to add it elsewhere in the array instead.

Basic Array updates - $push and $pop

$pop either removes the last item from an array, or the first if we set the value to -1



```
MongoDB> db.playlists.find({},{_id:0}).pretty()
{  "name" : "funky",
   "tracks" : [ {"artist" : "queen", "track" : "Liar" },
                {"artist" : "abba", "track" : "Chiquitita" },
                {"artist" : "AC/DC", "track" : "Hells Bells" }]
}

MongoDB> db.playlists.updateOne({name:"funky"},
{ $pop: {tracks: 1}})

MongoDB> db.playlists.find({},{_id:0}).pretty()
{  "name" : "funky",
   "tracks" : [{"artist" : "queen",  "track" : "Liar" },
               {"artist" : "abba",  "track" : "Chiquitita" }]
}

MongoDB> db.playlists.updateOne({name: "funky"},
{ $pop: {tracks: -1}})

MongoDB> db.playlists.find({},{_id:0}).pretty()
{   "name" : "funky",
   "tracks" : [{"artist" : "abba",  "track" : "Chiquitita" }]
}
```

$pop is used to remove things from an array - either the first or last element.

Basic Array updates - $pull and $pullAll

$pull and $pullAll selectively remove things from a list based on matching a given value or a query.

$pullAll allows to specify multiple specific values to remove

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

MongoDB> db.playlists.insertOne(
{name: "funky",
 tracks : [
  { artist:"queen",track:"Liar"},
  {artist:"abba",track:"Chiquitita"},
  { artist:"queen",track:"Under Pressure"},
  {artist:"AC/DC",track:"Hells Bells"},
 ]})

MongoDB> db.playlists.updateOne({name:"funky"},
{ $pull : { tracks : { artist: "queen" }}})

MongoDB> db.playlists.find({},{_id:0}).pretty()
{  "name" : "funky",
   "tracks" : [{"artist" : "abba",      "track" :
"Chiquitita" },
 {"artist" : "AC/DC",      "track" : "Hells Bells" }]}
```

For $pull you specify a value or query and all matching values are removed from the array.

Basic Array updates - $pull and $pullAll

Unlike $pull, with $pullAll you need to specify a list of values to remove.

Specify that list of values to remove as an array

```
MongoDB> db.playlists.drop()
MongoDB> db.playlists.insertOne(
{name: "funky",
 tracks : [
  { artist:"queen",track:"Liar"},
  {artist:"abba",track:"Chiquitita"},
  { artist:"queen",track:"Under Pressure"},
  {artist:"AC/DC",track:"Hells Bells"},
 ]})

MongoDB> db.playlists.updateOne({name:"funky"},
{ $pullAll : {"tracks" : [
 {artist:"abba",track:"Chiquitita"},
 {artist:"queen",track:"Under Pressure"}
]
}})

MongoDB> db.playlists.find({},{_id:0}).pretty()
{  "name" : "funky",
   "tracks" : [{"artist" : "queen",     "track" : "Liar" },
 {"artist" : "AC/DC",      "track" : "Hells Bells" }]}
```

For $pullAll you specify an array of values and all instances of any of them are removed.

Array Operators - $addToSet

$addToSet appends an element to an array only if it is not already present

$addToSet makes the array a unique list of values by not adding a value if the value is already there.
If there is already a duplicate this will not convert the array to a set, it will just not add another instance of the value.

Delete: deleteOne() and deleteMany()

deleteOne() and deleteMany() work the same way as updateOne() or updateMany()

Rather than taking a mutation - simply remove the document from the database

- Use deleteOne() and deleteMany() to remove documents
- Take query as an argument - do not forget this with deleteMany(), otherwise all documents will be deleted
- No commit safety net like with SQL
- deleteOne() will delete one document from the matching result set,
- deleteOne() will delete the first document it finds this depends on what index it selects to use and what order it chooses to traverse it in, which can depend on what previous queries have done too and therefore what is currently in cache. Assume you cannot predict what it will delete.

Updating, Locking and Concurrency

If two processes attempt to update the same document at the same time they are serialised

The conditions in the query must always match for the update to take place

In the example - if the two updates take place in parallel - the result is the same

```
MongoDB> db.lockdemo.insertOne({ _id: "Tom", votes: 0 } )

{"acknowledged" : true, "insertedId" : "Tom" }

MongoDB> db.lockdemo.updateOne({_id:"Tom",votes:0},
{$inc:{votes:1}})

{"acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1}

MongoDB> db.lockdemo.findOne({_id:"Tom"})

{"_id" : "Tom", "votes" : 1 }

MongoDB> db.lockdemo.updateOne({_id:"Tom",votes:0},
{$inc:{votes:1}})

{"acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0}

MongoDB> db.lockdemo.findOne({_id:"Tom"})

{ "_id" : "Tom", "votes" : 1 }

//This is True even if updates come in parallel from different clients
- all updates to a single document are serialized.
```

MongoDB performs individual updates on records serially, one update does not see another partially completed.
This means you can safely assume that the query conditions are true when making a change.
This does not affect reads, you can always read a record they are not serialised like writes.
Multiple writes can take place in parallel on a collection - this only affects individual documents.

Overwriting documents replaceOne()

replaceOne() takes a query and a replacement version of a document

Keeps only the **_id** field, all the others are replaced

Overwrites any values not in the submitted document

Best to avoid using it unless there is a very good reason to replace the whole document - use update and **$set** instead.

- replaceOne() overwrites document and only keeps _id field
- It is that this is ever required

Recap

Using Bulk writes vs. Single Writes has better network performance

find() returns us a cursor object which the shell then pulls from

In MongoDB there are various powerful update operators available

Quiz Time!

#1. When does a find() query get executed on the MongoDB server?

A	When a cursor is iterated	B	When you call the find() function	C	When the driver connects to the database
D	Every time we add a projection	E	Every time an index is created		

Answer in the next slide.

#1. When does a find() query get executed on the MongoDB server?

A	When a cursor is iterated	B	When you call the find() function	C	When the driver connects to the database
D	Every time we add a projection	E	Every time an index is created		

Answer: A
find() returns a cursor object rather than a document/s. The shell starts retrieving the first 20 results but by default find() on its own does not retrieve documents.
Calling find() does not return any values until you start retrieve data with the cursor.
The find() query does not have relationship with a connection pool or the driver connection.
The creation of a cursor, adding a projection, or creating an index do not execute the find query.

#2. Why is InsertMany() faster than multiple insertOne() operations?

A	Needs fewer writes to disk.
B	Reduces the network time.
C	Performs the writes as a single transaction.
D	Replicates to other servers faster.
E	Allows parallel processing of inserts in sharded clusters.

Answer in the next slide.

#2. Why is InsertMany() faster than multiple insertOne() operations?

A	Needs fewer writes to disk.	B	Reduces the network time.	C	Performs the writes as a single transaction.
D	Replicates to other servers faster.	E	Allows parallel processing of inserts in sharded clusters.		

Answer: B & E

Exercise Answers

Answer -Exercise: find, skip and limit

Write a find() to output only diary entries from "dug":

```
MongoDB> db.diaries.find({name:"dug"})
```

```
{"_id" : ObjectId("609ba812cf0c3aa225ce91de"), "name" : "dug", "day" : ISODate("2014-11-04T00:00:00Z"), "txt" : "went for a walk" }
```

```
{"_id" : ObjectId("609ba812cf0c3aa225ce91df"), "name" : "dug", "day" : ISODate("2014-11-06T00:00:00Z"), "txt" : "saw a squirrel" }
```

```
{"_id" : ObjectId("609ba812cf0c3aa225ce91e1"), "name" : "dug", "day" : ISODate("2014-11-09T00:00:00Z"), "txt" : "got a treat" }
```

Modify it to output the line below using skip, limit and a projection:

```
MongoDB> db.diaries.find({name:"dug"},{_id:0,day:0}).skip(1).limit(1)
```

```
{ name: "dug", txt: "saw a squirrel" }
```

Answer -Exercise: Range and Logic

How many documents in the grades collection have a student_id less than or equal to 65? Answer: 660

```
MongoDB> db.grades.find({student_id:{$lte:65}}).count()
```

How many documents in the inspections collection have result “Pass” or “Fail”? (Write two ways) Answer: 16609

```
MongoDB> db.inspections.find({$or:[{result:"Pass"},
{result:"Fail"}]}).count()
```

```
MongoDB> db.inspections.find({result:{$in:["Pass","Fail"]}}).count()
```

For a simple set of literals - $in is easier to read and allows more optimisation by the database - this is not a good use of $or even if it sounds like it

Answer -Exercise: $elemMatch

Use $elemMatch to find where an array element matches the query:

```
✓ db.restaurants.find({
    grades: {
       $elemMatch : {
         grade:"C",
         date:{$gt:ISODate("2013-12-31")}
        }
      }
})
```

Answer: 722 Restaurants to avoid.

```
MongoDB> db.restaurants.findOne()
{
        "_id" : ObjectId("5eb3d668b31de5d588f4292a"),
        "address" : {
                "building" : "2780",
                "coord" : [
                        -73.98241999999999,
                        40.579505
                ],
                "street" : "Stillwell Avenue",
                "zipcode" : "11224"
        },
        "borough" : "Brooklyn",
        "cuisine" : "American",
        "grades" : [
                {
                        "date" : ISODate("2014-06-10T00:00:00Z"),
                        "grade" : "A",
                        "score" : 5
                },
                {
                        "date" : ISODate("2013-06-05T00:00:00Z"),
                        "grade" : "A",
                        "score" : 7
                },
                {
                        "date" : ISODate("2012-04-13T00:00:00Z"),
                        "grade" : "A",
                        "score" : 12
                },
                {
                        "date" : ISODate("2011-10-12T00:00:00Z"),
                        "grade" : "C",
                        "score" : 12
                }
        ],
        "name" : "Riviera Caterer",
        "restaurant_id" : "40356018"
}
```

This time we want the Grade and the Date to be in the same element, it will not match this document.

```
db.restaurants.find({grades: {  $elemMatch : { grade:"C",date:{$gt: I
SODate("
2013-12-31")}}}}).count()
```

Only 722 restaurants to avoid.

Answer -Exercise: Combined Query

What company in the companies collection that has fewer than 200 employees has the most employees?

```
MongoDB> db.companies.find(
    {number_of_employees:{$lt:200}},
    {number_of_employees:1, name:1, _id: 0}
).sort({number_of_employees:-1}).limit(1)


{name: 'uShip',number_of_employees: 190}
```

Answer -Exercise: Updates (Part 1)

Exercise #1: Pass Inspections

In the collection, let's do a little data cleaning: Replace the "Completed" inspection result to use only "No Violation Issued" for those inspections. Update all the cases accordingly.

```
MongoDB> db.inspections.updateMany(
    {result:"Completed"},
    {$set:{result:"No Violation Issued"}}
)
```

Answer: 20 documents modified

Answer -Exercise: Updates (Part 2)

Exercise #2: Set fine value

For all inspections that fail, set a fine value of 100.

```
MongoDB> db.inspections.updateMany(
    {result:"Fail"},
    {$set:{fine_value:100}}
)
```

Answer: 1100 documents modified

Answer -Exercise: Updates (Part 3)

Exercise #3: Increase fine in ROSEDALE

Update all inspections done in the city of “ROSEDALE”, for failed inspections, raise the “fine” value by 150.

```
MongoDB> db.inspections.updateMany(
    {"address.city":"ROSEDALE",result:"Fail"},
    {$inc:{fine_value:150}}
)
```

Answer: 1 document modified