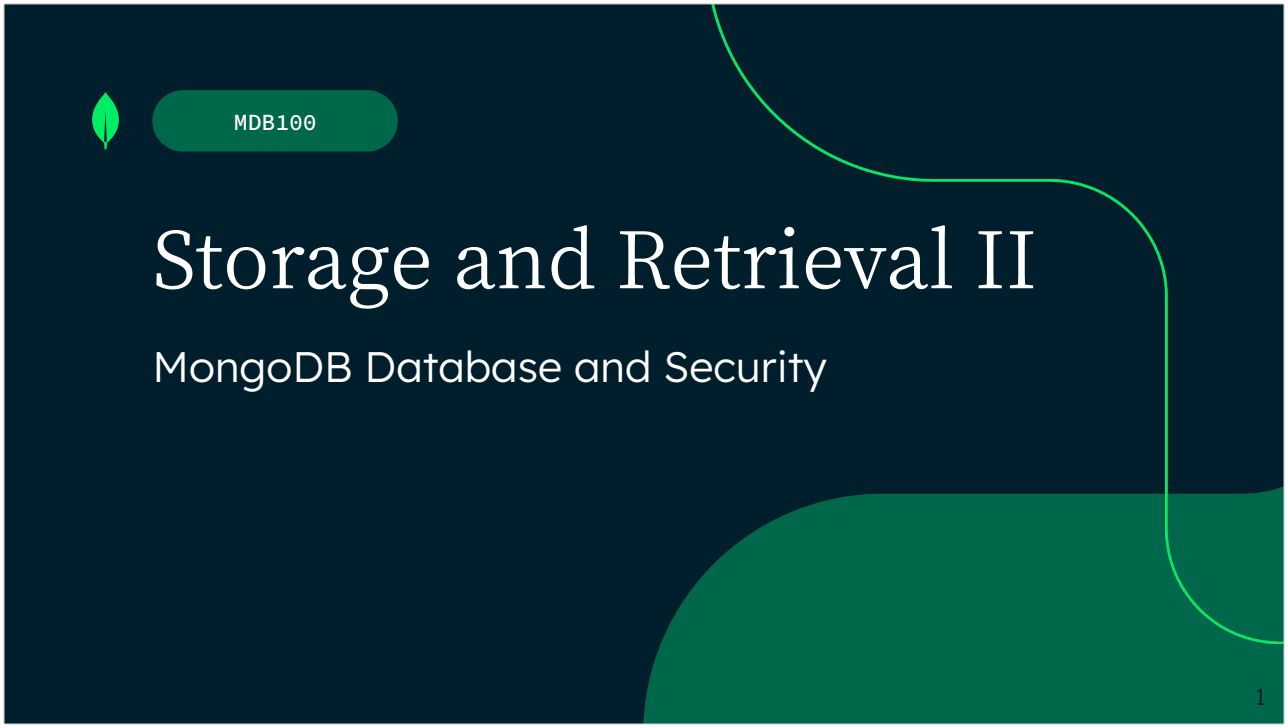
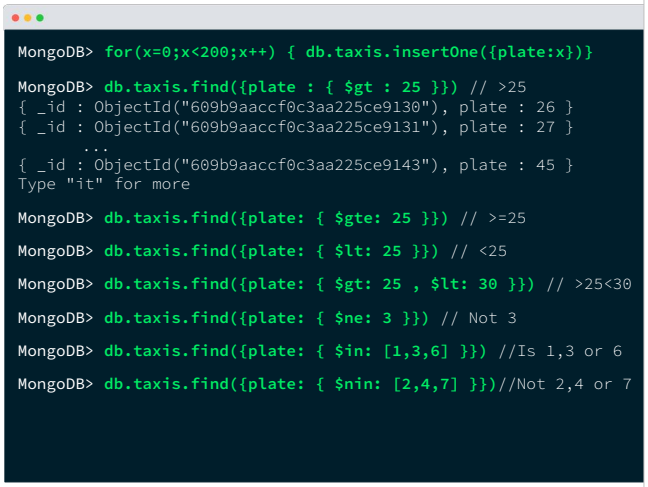


Release: 20250221

Range Operators

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)

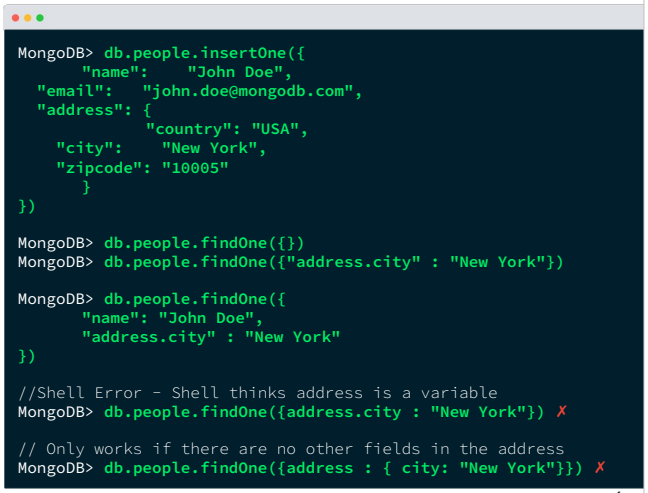
To change database type `use sample_training` in the shell
Answer at the end

Querying values in 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 thinks 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.

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, insertedId: ObjectId("...")}

//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`.

`$elemMatch` matches documents that contain an array field with at least one element that matches all the specified query criteria.

The $all operator is equivalent to an $and operator. $size and $elemMatch are array-specific. The functionality of an $any operator is already available with the $in operator.

ElemMatch

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

```
MongoDB> db.gamescore.insertMany([
{ "_id": "player1", "results": [{ "game": "pacman",
"score": 10 },
      { "game": "pong", "score": 5 }]
   },
{ "_id": "player2", "results": [{ "game": "pacman",
"score": 5 },
      { "product": "pong", "score": 7 }]
   }
])

{acknowledged:true }

MongoDB> db.gamescore.find({
 results: { $elemMatch:{ game: "pacman", score: {$gt:5}}
}})

{ _id : 'player1', results: [...] }
```

Expressive Queries

$expr can query with Aggregations

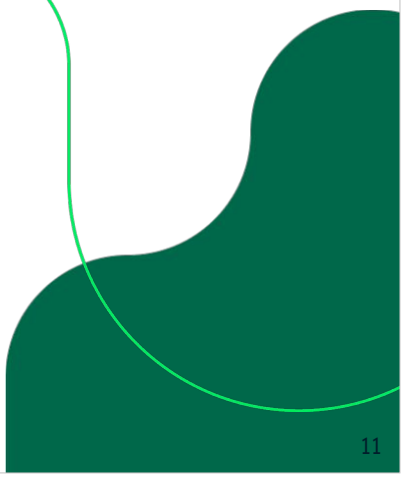
$expr can match ANY computable function in the data.

$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

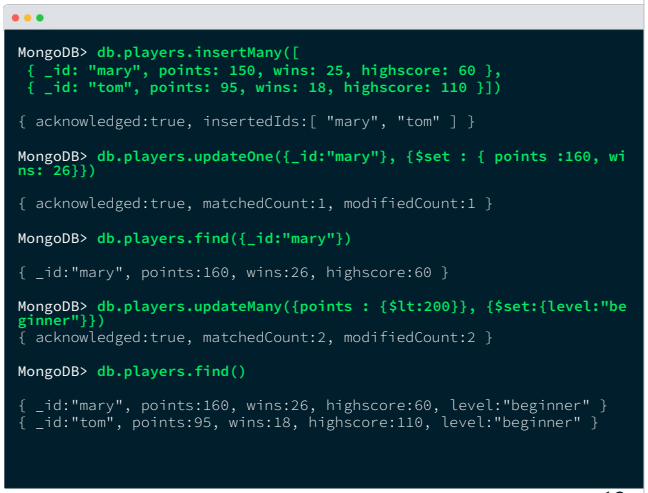


Updating Documents

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. We can handle this problem using transactions.
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 an empty string, or 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":""}` expression doesn't actually matter as it would remove the field anyways. You can also use 1 or true:

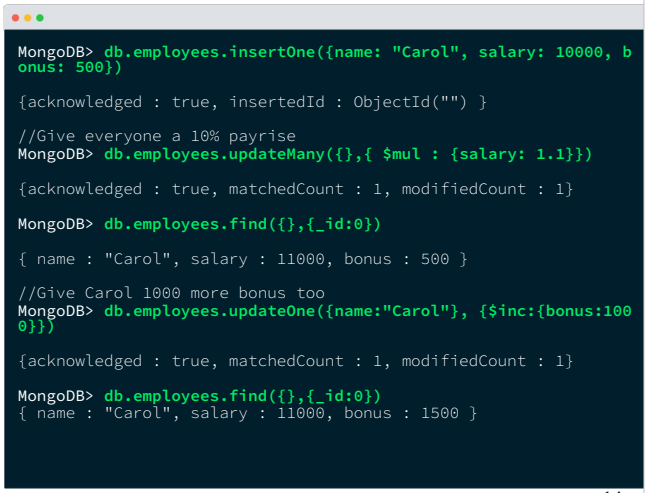
```
$unset: {"Singer":1}
$unset: {"Singer":true}
```

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": 9000}})

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

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

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

MongoDB> db.gamescores.find({})
{ _id : ObjectId("..."), 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.

Exercise - Updating records

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.

The WiredTiger Storage engine uses Multi-Version Concurrency Control (MVCC) to achieve this.

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.

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 a document and only keeps the _id field
Rarely required.

Quiz Time!

#1. What two arguments does an update operation need?

A	query	B	collection	C	index
D	document	E	mutation		

Answer in the next slide.

#1. What two arguments does an update operation need?

A query

B collection

C index

D document

E mutation

Regardless of if its an updateOne or an updateMany - the syntax has the following structure "...updateOne(query, mutation)" the first argument being what documents to match on based on query and the second what update operator to use, any arguments that follow are optional the query itself can also be left blank to tell the engine to match on the first document or all documents depending on the operation.

#2. How are concurrent write processes handled?

A	In parallel	B	Sequentially	C	As a single atomic transaction
D	As a bulk write	E	Multi-Version Concurrency Control		

Answer in the next slide.

#2. How are concurrent write processes handled?

A	In parallel	B	Sequentially	C	As a single atomic transaction
D	As a bulk write	E	Multi-Version Concurrency Control		

Locking works similar to how SQL manages locks. Each process tries to acquire a lock on the document and the one that is able to first maintains the lock until the process completes its work. The lock releases and the process in wait will acquire a lock and attempt to do what it needs to do, hence sequentially. Each process also technically revisions the document and and reads that come in for that document before the update is completed is able to read the old version of the document and the new version afterwards as managed by the MVCC of wiredTiger. Concurrent processes are never handled in parallel, nor are they treated a single atomic operations with the notion of automatic rollbacks.

Recap

In MongoDB there are various powerful operators available to query and update documents

Updates are atomic on the document level

Exercise Answers

Answer -Exercise: Range and Logic

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

```
db.grades.countDocuments({student_id:{$lte:65}})
```

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

```
db.inspections.countDocuments({$or:[{result:"Pass"},{result:"Fail"}]})
```

```
db.inspections.countDocuments({result:{$in:["Pass","Fail"]}})
```


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: 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