



# Storage and Retrieval 2

DF100 - MongoDB Developer Fundamentals

# Topics we cover

---

- Array Updates
- Expressive Updates
- Upserts
- FindOneAndUpdate

# Updating Arrays

---

In the next set of slides we look at different ways to update specific members of arrays.

Imagine our database contains information about the number of hours a person "Tom" will work so the array "hrs" of 5 members represents days of the week, each is how many hours they need to work that day.

This example is kept very minimalist for clarity.

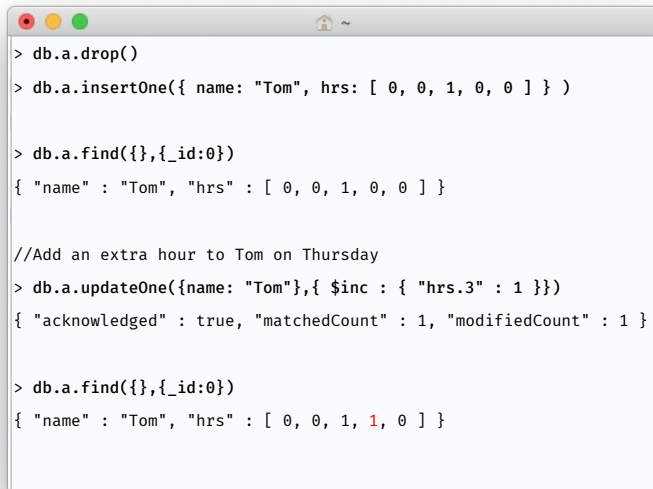


# Arrays - Update Specific Element

We can update a specific array element by using a numeric index

These start at 0

Arrays grow as required.



```
> db.a.drop()
> db.a.insertOne({ name: "Tom", hrs: [ 0, 0, 1, 0, 0 ] })

> db.a.find({}, {_id:0})
{ "name" : "Tom", "hrs" : [ 0, 0, 1, 0, 0 ] }

//Add an extra hour to Tom on Thursday
> db.a.updateOne({name: "Tom"},{ $inc : { "hrs.3" : 1 }})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.a.find({}, {_id:0})
{ "name" : "Tom", "hrs" : [ 0, 0, 1, 1, 0 ] }
```

- Array elements can be updated by using numeric index e.g.
  - `db.a.updateOne({name:"Tom"},{ $inc : { "hrs.3" : 1 }})`
- Arrays are zero indexed in MongoDB
- If we explicitly set a value which is higher than the length of the array, the array will be extended to the required length and all the new elements except the one we are setting will be set to null.

# Arrays - Update Matched Element

We can use **\$** as a placeholder for the first position matched by a query

In this case find array with a value less than 1

Ignore the fact we are also querying by primary key

```
> db.a.drop()
//Tom works one hour on wednesday
> db.a.insertOne({ name: "Tom", hrs: [ 0, 0, 1, 0, 0 ] })

//Find the first day tom has no hours and add two to that day
> db.a.updateOne({name:"Tom",hrs:{$lt:1}},{ $inc: { "hrs.$": 2 }})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

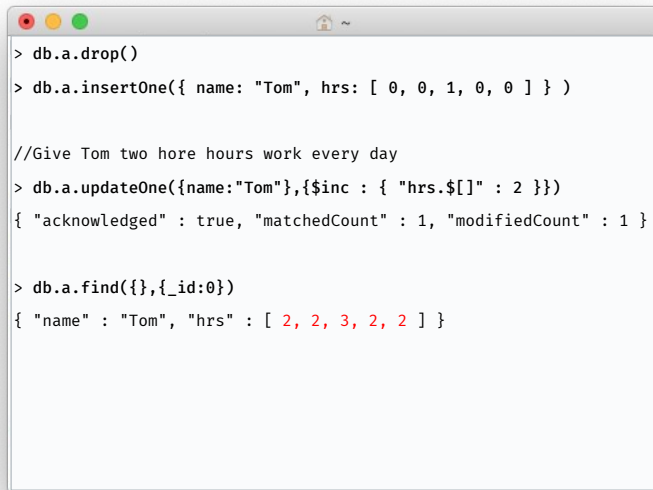
> db.a.find({},{_id:0})
{ "name" : "Tom", "hrs" : [ 2, 0, 1, 0, 0 ] }
```

- \$ can be used as a placeholder for first position matched by query e.g.
  - `db.a.updateOne({_id:"Tom",hrs:{$lt:1}},{ $inc : { "hrs.$" : 2 }})`

# Arrays - Update All Elements

We can update every element of an array by specifying it with `$[ ]`

In this example we are adding 2 to every entry in hours for every document that matches our query.



```
> db.a.drop()
> db.a.insertOne({ name: "Tom", hrs: [ 0, 0, 1, 0, 0 ] })

//Give Tom two hore hours work every day
> db.a.updateOne({name:"Tom"},{$inc : { "hrs.$[]" : 2 }})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.a.find({}, {_id:0})
{ "name" : "Tom", "hrs" : [ 2, 2, 3, 2, 2 ] }
```

- `$[ ]` changes all array elements
- Was added in MDB 3.6
- Not widely used

# Arrays - Update All Matched Elements

Query to find documents is not used to decide what elements to change

Separate `arrayFilter(s)` apply update to matching array elements

This example says "add 2 to everything < 1 in hrs"

```
> db.a.drop()
> db.a.insertOne({ name: "Tom", hrs: [ 0, 0, 1, 0, 0 ] })

//Find a week where tom has a day with no hours (query)
// for each day Tom has no hours and add 2 to those days
// (arrayFilter) - assume there might be multiple records for Tom
// so we can cannot use JUST arrayfilters.

> db.a.updateOne({name:"Tom",hrs:{$lt:1}},
  {$inc : { "hrs.$[nohrs]" : 2 }},
  {"arrayFilters": [ { "nohrs":{"$lt":1}}]})

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

> db.a.find({},{_id:0})
{ "name" : "Tom", "hrs" : [ 2, 2, 1, 2, 2 ] }
```

- Array filters allow us to select specific members of an array to modify by defining addition match criteria/queries.
- They are separate from the `find()` part of the query so we can `find()` with one query but then decide how to update the array based on different criteria
- If we do include some of the filter criteria in the `find()` part as we have done here - then we can avoid the computation of even trying to update records where the filter won't match.
- We can define multiple named array filters in our update - we called this one **nohrs**, nohrs is called an identifier
- In the filter the identifier is evaluated against each array element and those that match are updated.
- Here we say If the element is less than one then update it
- If the array elements are objects - we can dereference them in the filter with dots - e.g in an array of items on a purchase receipt `{ "arrayFilters": [ { "item.type": "socks", "item.price" : {$gt:5} } ] }`

# Arrays and \$each

When we use \$push to add to an array that value is added as the last element.

In this case we now have an array inside an array.

However, we wanted to add just the members.

```
> db.a.drop()
//Set Toms hours for Monday to Wednesday
> db.a.insertOne({ "name" : "Tom", "hrs" : [ 4, 1, 3 ] })

//Add hours for Thursday and Friday Incorrectly
> db.a.updateOne({name:"Tom"}, {$push:{hrs:[2,9]}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.a.find({},{_id:0})
{ "name" : "Tom", "hrs" : [ 4, 1, 3, [ 2, 9 ] ] }
```

In the example, \$push does not give the desired effect - the array itself is added to the existing array, not the individual elements



# Arrays and \$each

Using \$each pushes the elements separately

Also works with \$addToSet

```
> db.a.drop()
> db.a.insertOne({ "name" : "Tom", "hrs" : [ 4, 1, 3 ] })

//Add Tom's hours for Thursday and Friday Correctly
> db.a.updateOne({name:"Tom"}, {$push:{hrs:{ $each : [2,9]}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

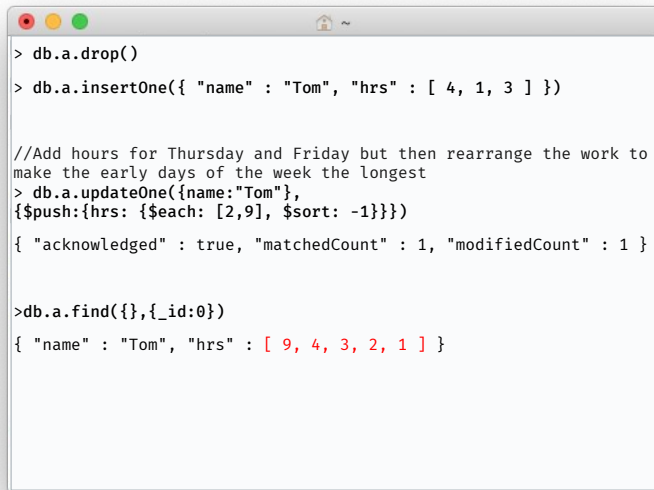
> db.a.find({}, {_id:0})
{ "name" : "Tom", "hrs" : [ 4, 1, 3, 2, 9 ] }
```

- Using \$each adds the element separately to the array
- \$addToSet would also achieve this

# Arrays and \$each

As we push them in we can also sort the array

If we try to do this client side we will likely get race conditions

A terminal window with a light gray background and a title bar with red, yellow, and green window control buttons. The terminal shows a series of MongoDB commands and their output. The commands are: `db.a.drop()`, `db.a.insertOne({ "name" : "Tom", "hrs" : [ 4, 1, 3 ] })`, a comment `//Add hours for Thursday and Friday but then rearrange the work to make the early days of the week the longest`, `db.a.updateOne({name:"Tom"}, {$push:{hrs: {$each: [2,9], $sort: -1}}})`, and `db.a.find({}, {_id:0})`. The output for the update command is `{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }` and the output for the find command is `{ "name" : "Tom", "hrs" : [ 9, 4, 3, 2, 1 ] }`.

```
> db.a.drop()
> db.a.insertOne({ "name" : "Tom", "hrs" : [ 4, 1, 3 ] })

//Add hours for Thursday and Friday but then rearrange the work to
//make the early days of the week the longest
> db.a.updateOne({name:"Tom"},
{$push:{hrs: {$each: [2,9], $sort: -1}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

>db.a.find({}, {_id:0})
{ "name" : "Tom", "hrs" : [ 9, 4, 3, 2, 1 ] }
```

Elements can also be sorted as they are pushed but doing this client side will likely get a race condition

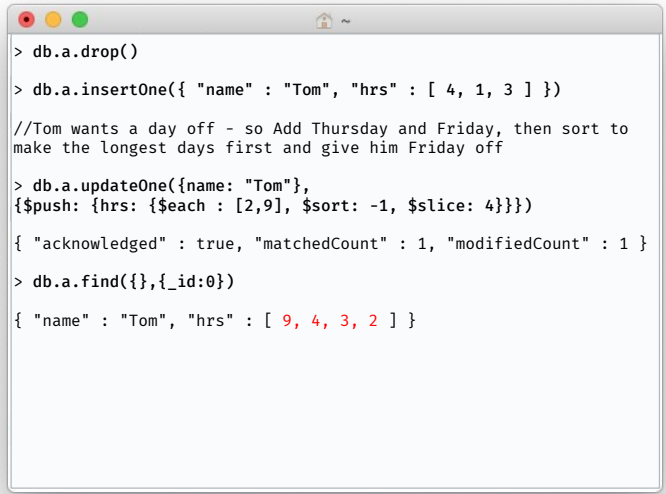
A race condition is when performing simultaneous actions can result in undesired effects

# Arrays and \$each

We can also sort and keep the Top(or bottom) N

This is an example of a design pattern.

Used for High/Low lists – high scores, top 10 temperatures etc.



```
> db.a.drop()

> db.a.insertOne({ "name" : "Tom", "hrs" : [ 4, 1, 3 ] })

//Tom wants a day off - so Add Thursday and Friday, then sort to make the longest days first and give him Friday off

> db.a.updateOne({name: "Tom"},
{$push: {hrs: {$each : [2,9], $sort: -1, $slice: 4}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.a.find({}, {_id:0})
{ "name" : "Tom", "hrs" : [ 9, 4, 3, 2 ] }
```

\$slice allows us to keep the top (or bottom) N number of elements

# Expressive Updates

Expressive updates use aggregation pipeline expressions to define what fields to \$set and their new values.

Updates can be based on other values in the document or conditions calculated from them.

Example: Add an area field to all our rectangular records to make it easy to search by size. If we add a field like this we can build an index on it.

```
db.shapes.updateMany(  
  { shapename: {$in : ["rectangle","square"]} },  
  [ { $set: { area: { $multiply:[ "$width","$height" ] } } } ]  
)
```

- Updates can also be performed using an aggregation pipelines expression - we cover them later in the course.
- This can be used to update a document based on other values in the document e.g. if a=1 then add 2 to b else add c to b
- The update operation is an array containing a pipeline that outputs the new changes
- The update mutation is an array of pipeline stages, hence the square brackets.

## Exercise – Array updates

---

Again, use the **sample\_training** database in the Mongo shell.

In the **grades** collection – if anyone got >90% for any homework, reduce their score by 10.

In the **grades** collection add a new field containing their mean score in each class.

Drop each student's worst score in the **grades** collection. You need to use two updates to do this.



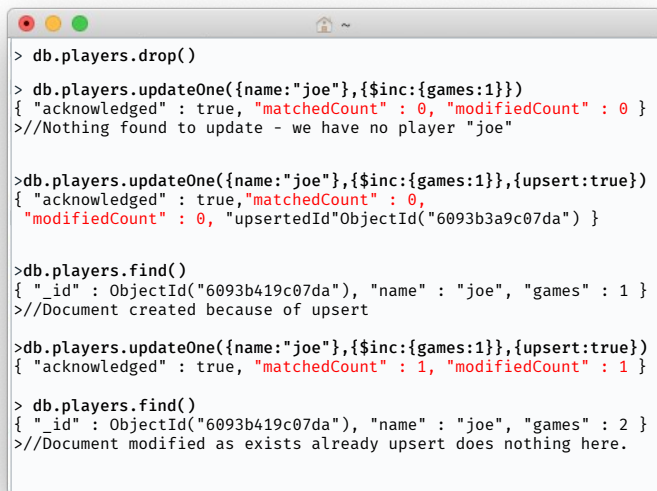
**Hint:** Use the **\$avg** operator for calculating the mean score.  
How do you get the Students worst score to a known place in the array?

# Upsert

Most MongoDB operations that update also allow the flag "upsert:true"

Upsert "Inserts" a new document if none are found to update.

Values in both the Query and Update are used to create the new record.



```
> db.players.drop()

> db.players.updateOne({name:"joe"},{$inc:{games:1}})
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
> //Nothing found to update - we have no player "joe"

> db.players.updateOne({name:"joe"},{$inc:{games:1}}, {upsert:true})
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0, "upsertedId" : ObjectId("6093b3a9c07da") }

> db.players.find()
{ "_id" : ObjectId("6093b419c07da"), "name" : "joe", "games" : 1 }
> //Document created because of upsert

> db.players.updateOne({name:"joe"},{$inc:{games:1}}, {upsert:true})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.players.find()
{ "_id" : ObjectId("6093b419c07da"), "name" : "joe", "games" : 2 }
> //Document modified as exists already upsert does nothing here.
```

- Upserts are a useful feature that will be covered more later
- If a document isn't found to update then one that matches your criteria is created
- Passed as an option to updateOne() and updateMany() and other update operations
- Not atomic between find() and insert() - it's possible for two simultaneously run to both do an insert.
  - If this is an issue then supplying \_id using \$setOnInsert update modifier helps
  - unique constraints help too.
- Useful in a few design patterns and simplifies code versus a call to update then to insert.

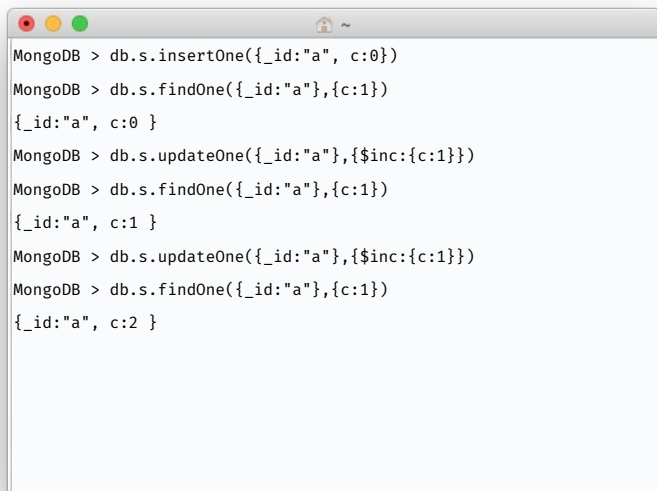
# findOneAndUpdate()

To understand findOneAndUpdate(), we must first understand updateOne().

In updateOne the find and the change are atomic

However updateOne doesn't return the updated document.

Imagine getting the next one-up number from a sequence



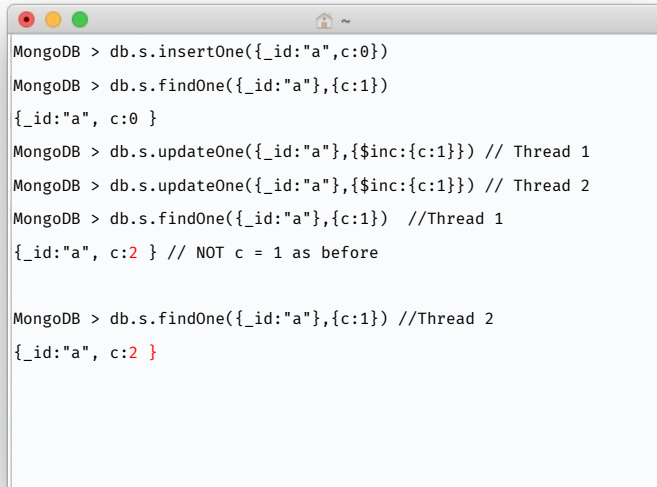
```
MongoDB > db.s.insertOne({_id:"a", c:0})
MongoDB > db.s.findOne({_id:"a"},{c:1})
{_id:"a", c:0 }
MongoDB > db.s.updateOne({_id:"a"},{$inc:{c:1}})
MongoDB > db.s.findOne({_id:"a"},{c:1})
{_id:"a", c:1 }
MongoDB > db.s.updateOne({_id:"a"},{$inc:{c:1}})
MongoDB > db.s.findOne({_id:"a"},{c:1})
{_id:"a", c:2 }
```

If we wanted to update a value and fetch it, to create a sequence say for a "Customer number" or "Invoice NUmber" we could do it like this - and in a single thread this is fine. If there is no projection applied in the findOne() commands, the result would still be the same as there is only one field apart from the \_id field.

# findOneAndUpdate()

Now Imagine 2 parallel processes / threads running updateOne() and findOne() on the same document.

What if they interleave the calls!



```
MongoDB > db.s.insertOne({_id:"a",c:0})
MongoDB > db.s.findOne({_id:"a"},{c:1})
{_id:"a", c:0 }
MongoDB > db.s.updateOne({_id:"a"},{$inc:{c:1}}) // Thread 1
MongoDB > db.s.updateOne({_id:"a"},{$inc:{c:1}}) // Thread 2
MongoDB > db.s.findOne({_id:"a"},{c:1}) //Thread 1
{_id:"a", c:2 } // NOT c = 1 as before

MongoDB > db.s.findOne({_id:"a"},{c:1}) //Thread 2
{_id:"a", c:2 }
```

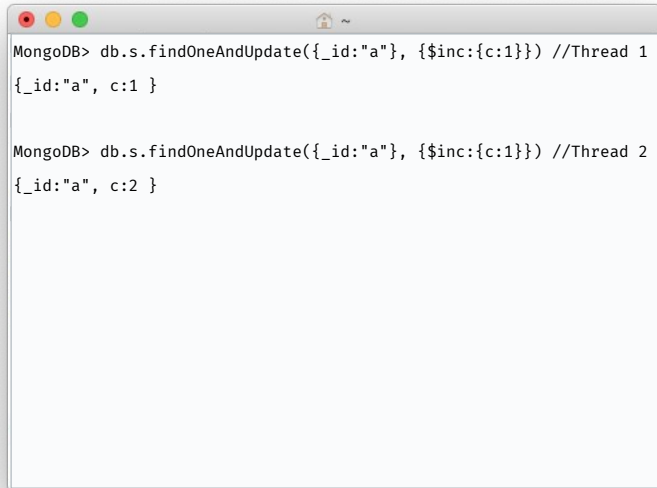
If we extend this to multiple threads then we can have a race condition where we update twice then fetch twice.

It would be fine if we don't provide the projection in findOne() commands as there is only one field other than the \_id. It would give us the same result.



# findOneAndUpdate()

findOneAndUpdate() fixes this by making find, modify, and fetch atomic



```
MongoDB> db.s.findOneAndUpdate({_id:"a"}, {$inc:{c:1}}) //Thread 1
{_id:"a", c:1 }

MongoDB> db.s.findOneAndUpdate({_id:"a"}, {$inc:{c:1}}) //Thread 2
{_id:"a", c:2 }
```

findOneAndUpdate() solves issues from previous slide

With findOneAndUpdate the single, atomic command finds, modifies and returns the document. It can return either the document before or after the change and you can apply things like sorting and projection to control what it finds and modified.

It only affects one document and also will work with upsert.

# Recap

---

- There are a number of ways to update specific members of an Array
- Expressive Updates allow us to compute new values for fields inside the database.
- Upsert automatically creates new record where none is available to edit.
- FindOneAndUpdate atomically returns the version of the document you edited as part of the edit.





## Answers

# Answers - Exercise - Array updates

In the grades collection - if anyone got >90% for any homework, reduce their score by 10.

```
db.grades.updateMany({
  scores:{$elemMatch:{type:"homework", score:{$gt:90}}}},
{$inc: {"scores.$[filter].score":-10}},
{ arrayFilters: [{ "filter.type":"homework",
  "filter.score": {$gt:90}}]})
```

Note we need to use \$elemMatch



# Answers - Exercise - Array updates

In the grades collection add a new field containing their the mean score in each class.

```
db.grades.updateMany(
  {},
  [{ $set: { average : { $avg: "$scores.score" }}}]
)
```



# Answers - Exercise - Array updates

Drop each student's worst score. You may use multiple updates to do this.

```
db.grades.updateMany(
  {},
  {
    $push: {
      scores: {
        $each: [],
        $sort: { score: 1 }
      }
    }
  }
)
```

```
db.grades.updateMany(
  {},
  {
    $pop: { scores: -1 }
  }
)
```

