DF300

Internals for developers

Design Skills and Advanced Features

Release: 20230414

Topics we cover

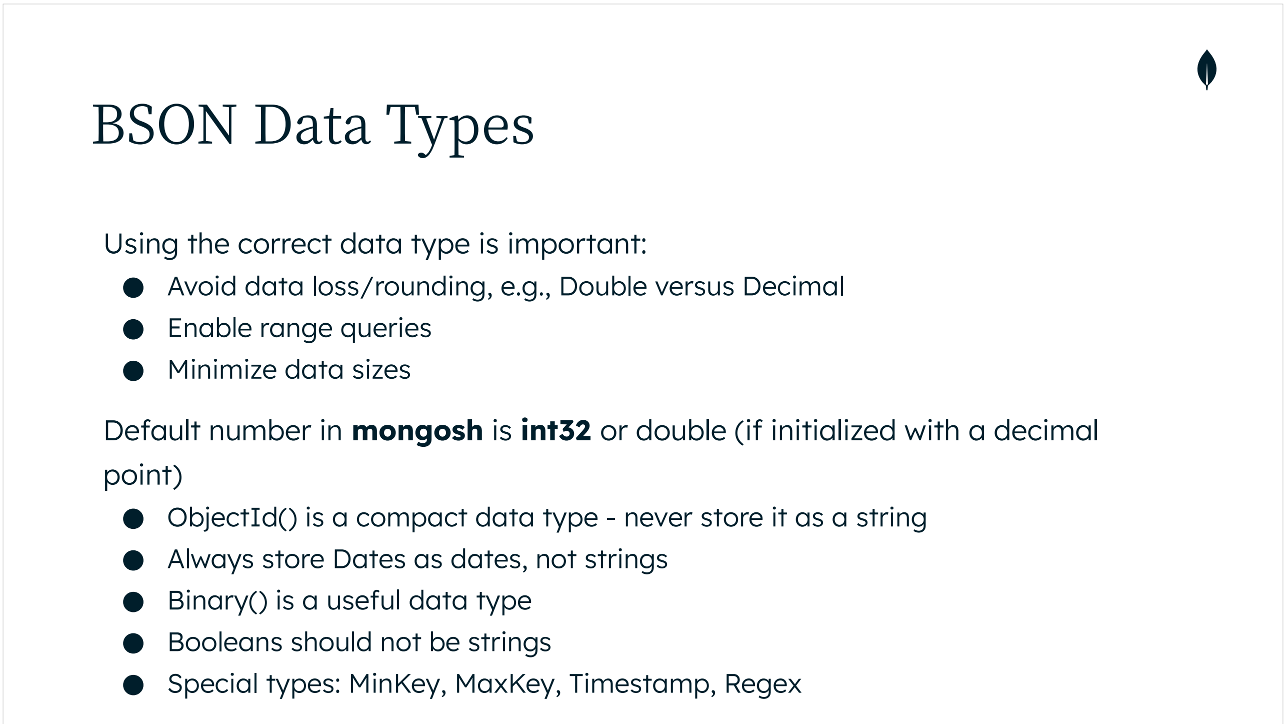
BSON data types and Null handling

Collation and sort ordering

Type bracketing

Sorting by objects and arrays

Locking



BSON Data Types

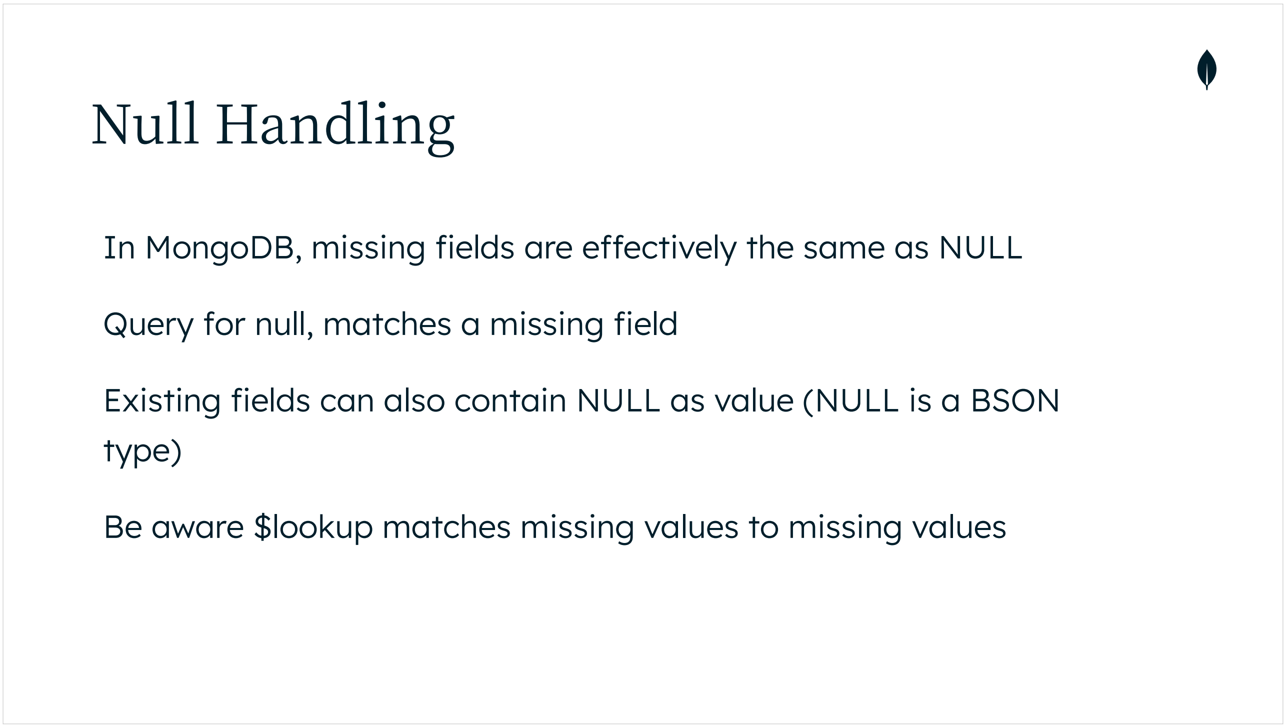
Using the correct data type is important:

- Avoid data loss/rounding, e.g., Double versus Decimal
- Enable range queries
- Minimize data sizes

Default number in **mongosh** is **int32** or double (if initialized with a decimal point)

- ObjectId() is a compact data type - never store it as a string
- Always store Dates as dates, not strings
- Binary() is a useful data type
- Booleans should not be strings
- Special types: MinKey, MaxKey, Timestamp, Regex

- BSON data is stored in a typed binary form, unlike JSON, which is just text, and types need to be inferred.
- Ensure if using an untyped format like JSON data is converted to an appropriate data type.
- MongoDB tools have MongoDB Extended JSON - a form of JSON that specifies types.
- Using the correct data types makes the data more compact, less prone to rounding errors, sort correctly, and allows correct querying.
- In mongosh - the default number type is int32. You can explicitly use Long or Double as needed
- Use the Binary data type rather than text encoded binary data
- There are a few specialized types like minKey - which is smaller than everything both in sorting and in size
- BSON is on version 1.1 - we only ever added one thing to it - Decimal 128 (although we did allocate Binary subtype 6 for encrypted data it didn't change the format)



Null Handling

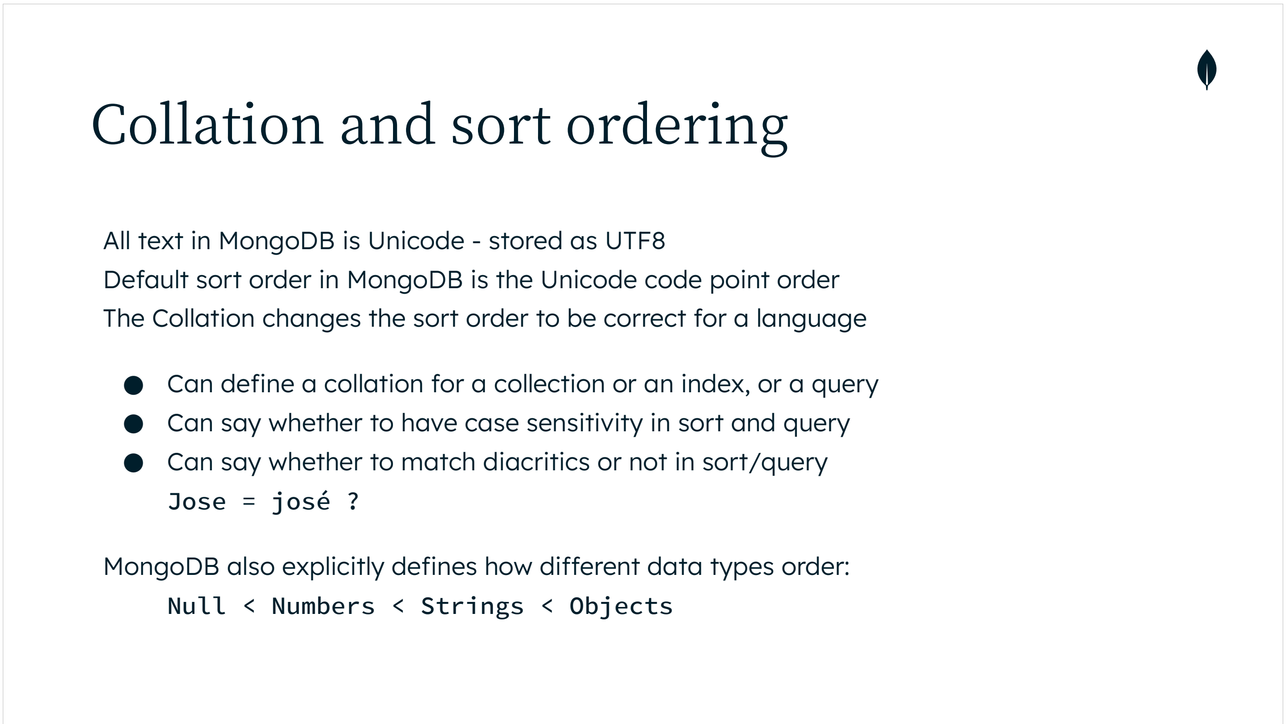
In MongoDB, missing fields are effectively the same as NULL

Query for null, matches a missing field

Existing fields can also contain NULL as value (NULL is a BSON type)

Be aware $lookup matches missing values to missing values

- Null is a specific data type in MongoDB with one value
- However, MongoDB treats the absence of a value for a given key as an explicit null
- When reading, if you ask for the value of a non-existent field, you get a null.
- If you query where field = null, you get documents where the field does not exist - you can use $exists, but that information isn't indexed.
- Projecting a non-existent field doesn’t show us null. It doesn’t show us the field altogether.



Collation and sort ordering

All text in MongoDB is Unicode - stored as UTF8
Default sort order in MongoDB is the Unicode code point order
The Collation changes the sort order to be correct for a language

- Can define a collation for a collection or an index, or a query
- Can say whether to have case sensitivity in sort and query
- Can say whether to match diacritics or not in sort/query

```
Jose = josé ?
```

MongoDB also explicitly defines how different data types order:

```
Null < Numbers < Strings < Objects
```

- Basic sorting in MongoDB is by the Unicode code point order
- It doesn't understand language and culture semantics for sorting, like in Germany, where dictionaries and phonebooks sort in a different order.
- In MongoDB, you can use collation to make sorts and indexes (and finds) work with case and or diacritic (characters over letters) insensitivity.
- Also, in MongoDB - as a field can hold different types - there is a defined order that different types sort when compared to each other.

Sorting by Objects

Objects can be compared to each other

When objects are compared, the field order matters

- Some programming languages don't preserve the field order by default
- Objects are compared field-by-field
- If field names differ, then the order is by first changed field name
- If values differ, then the order is by value

```
{ a: 1 } < { a:1, b: 1} < { b:1, a:1}
```

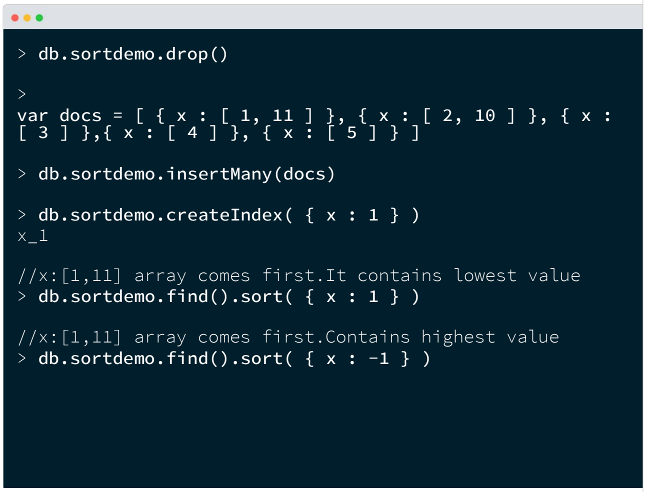
- You can compare two objects and say one is greater or less than another
- This is done by treating them as an ordered list of fields. If the field names don't match, then the order is based on the field name; otherwise, the value
- Understanding how this works allows you to index and compare objects rather than members, and this can be very useful.

Sorting by Arrays

When querying an array, semantics refers to the whole array or any member

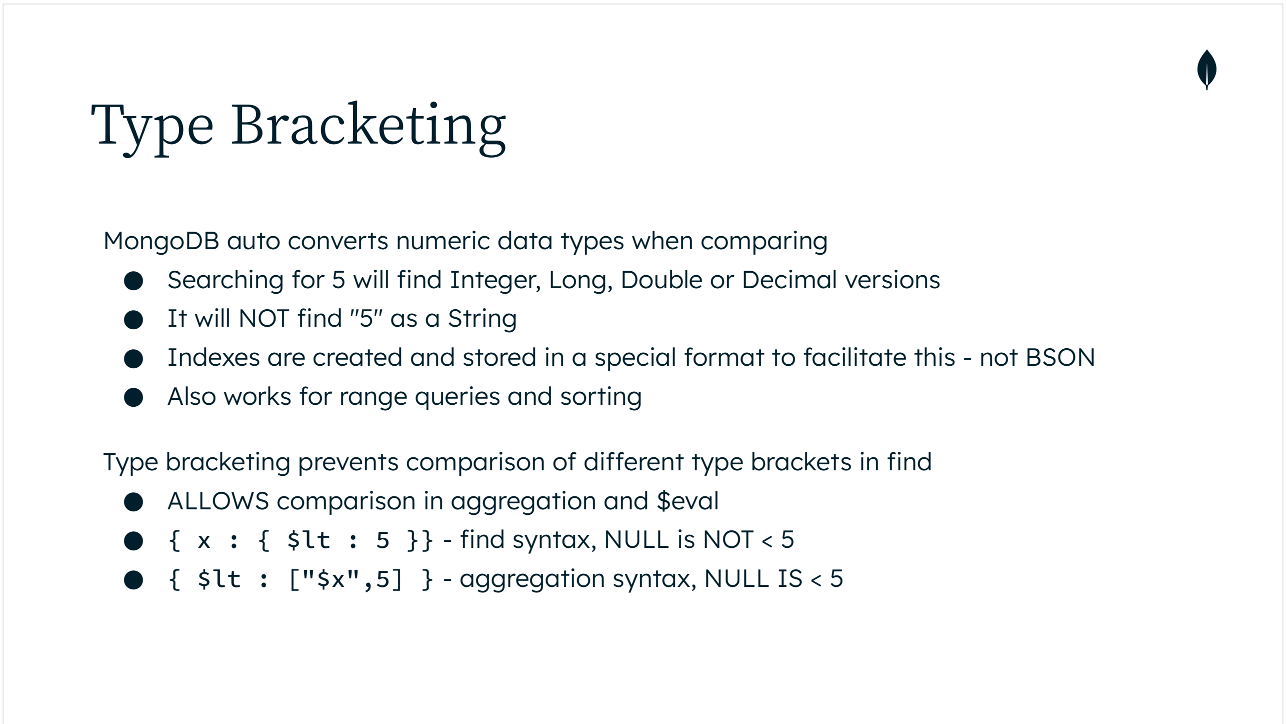
So { $gt : 5 } is true where any array member is greater than 5

When sorting, the highest or the lowest value in the array is used as the sort value



```
> db.sortdemo.drop()

>
var docs = [ { x : [ 1, 11 ] }, { x : [ 2, 10 ] }, { x :
[ 3 ] },{ x : [ 4 ] }, { x : [ 5 ] } ]

> db.sortdemo.insertMany(docs)

> db.sortdemo.createIndex( { x : 1 } )
x_1

//x:[1,11] array comes first.It contains lowest value
> db.sortdemo.find().sort( { x : 1 } )

//x:[1,11] array comes first.Contains highest value
> db.sortdemo.find().sort( { x : -1 } )
```

- When you query an array - remember you match either one member or the entire array.
- When you sort by a field that contains an array - if sorting by increasing size, the lowest value in the array is used, otherwise the highest - the sorting is done on the first eligible member of the array, not the whole array.
- If interested - Binary data is sorted by Length, then subtype, then content.



Type Bracketing

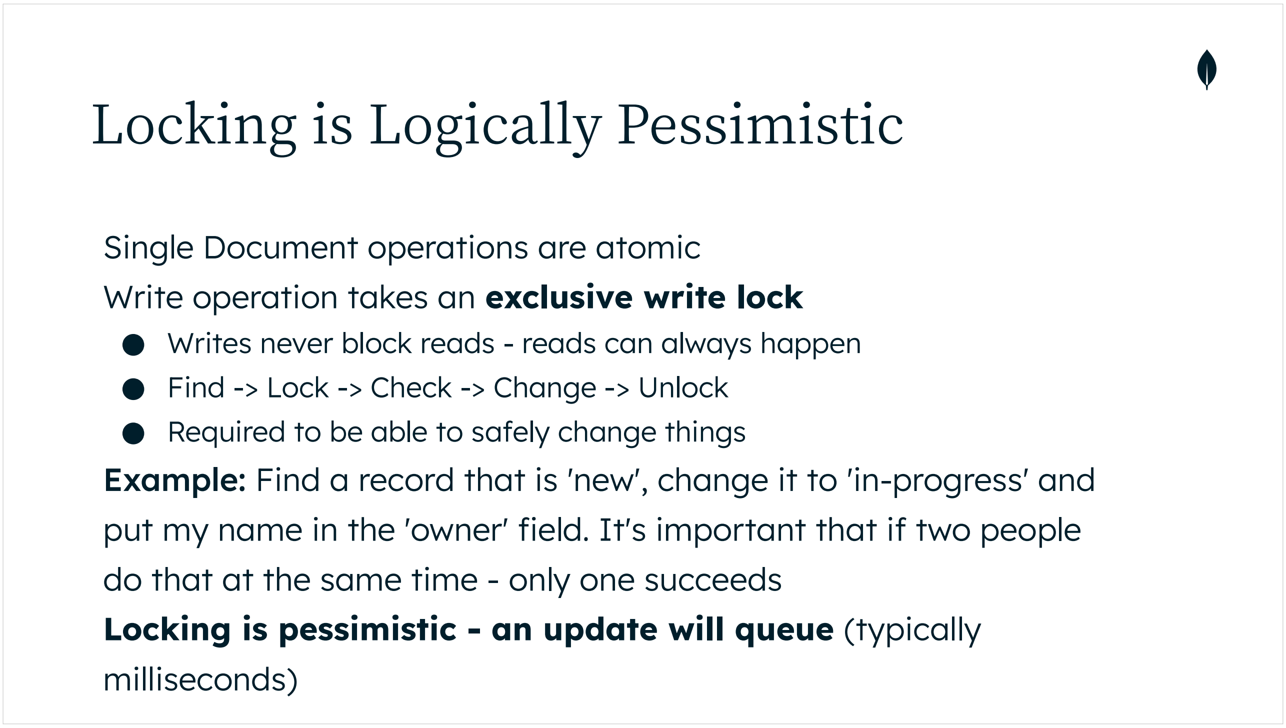
MongoDB auto converts numeric data types when comparing

- Searching for 5 will find Integer, Long, Double or Decimal versions
- It will NOT find "5" as a String
- Indexes are created and stored in a special format to facilitate this - not BSON
- Also works for range queries and sorting

Type bracketing prevents comparison of different type brackets in find

- ALLOWS comparison in aggregation and $eval
- `{ x : { $lt : 5 }}` - find syntax, NULL is NOT < 5
- `{ $lt : ["$x",5] }` - aggregation syntax, NULL IS < 5

- MongoDB is smart when it comes to comparing numbers
- It will compare numbers of different types correctly - but only numeric types
- It also considers type when comparing in find() but not in an explicit expression.
- For more details about type bracketing check this link: https://docs.mongodb.com/v3.2/reference/method/db.collection.find/#type-bracketing



Locking is Logically Pessimistic

Single Document operations are atomic

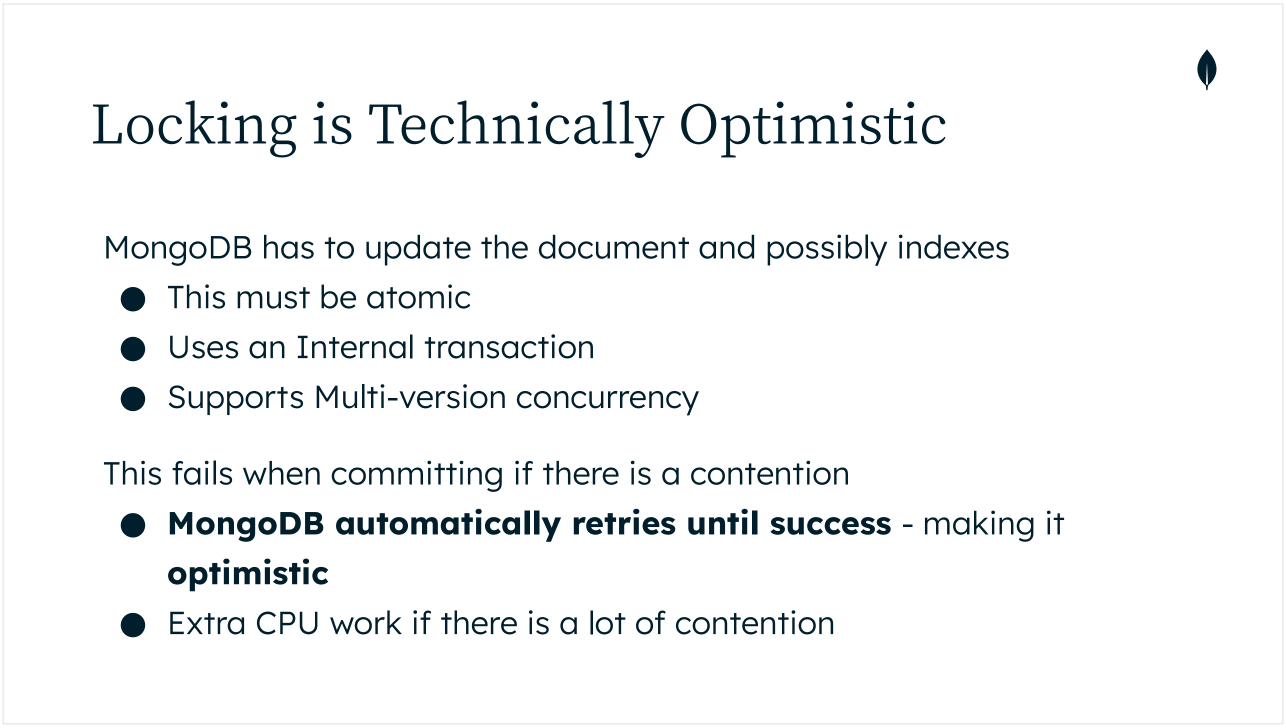
Write operation takes an **exclusive write lock**

- Writes never block reads - reads can always happen
- Find -> Lock -> Check -> Change -> Unlock
- Required to be able to safely change things

Example: Find a record that is 'new', change it to 'in-progress' and put my name in the 'owner' field. It's important that if two people do that at the same time - only one succeeds

Locking is pessimistic - an update will queue (typically milliseconds)

- MongoDB binds parts of operations together to ensure consistency
- Writes - for each document find() and update() are inside an exclusive lock
- findOneAndUpdate - for each document find(), update() and read() are effectively inside an exclusive lock
- This is required to be able to consistently update things
- Locking is seen as pessimistic - if it's locked, your process will wait until unlocked (typically very quick)



Locking is Technically Optimistic

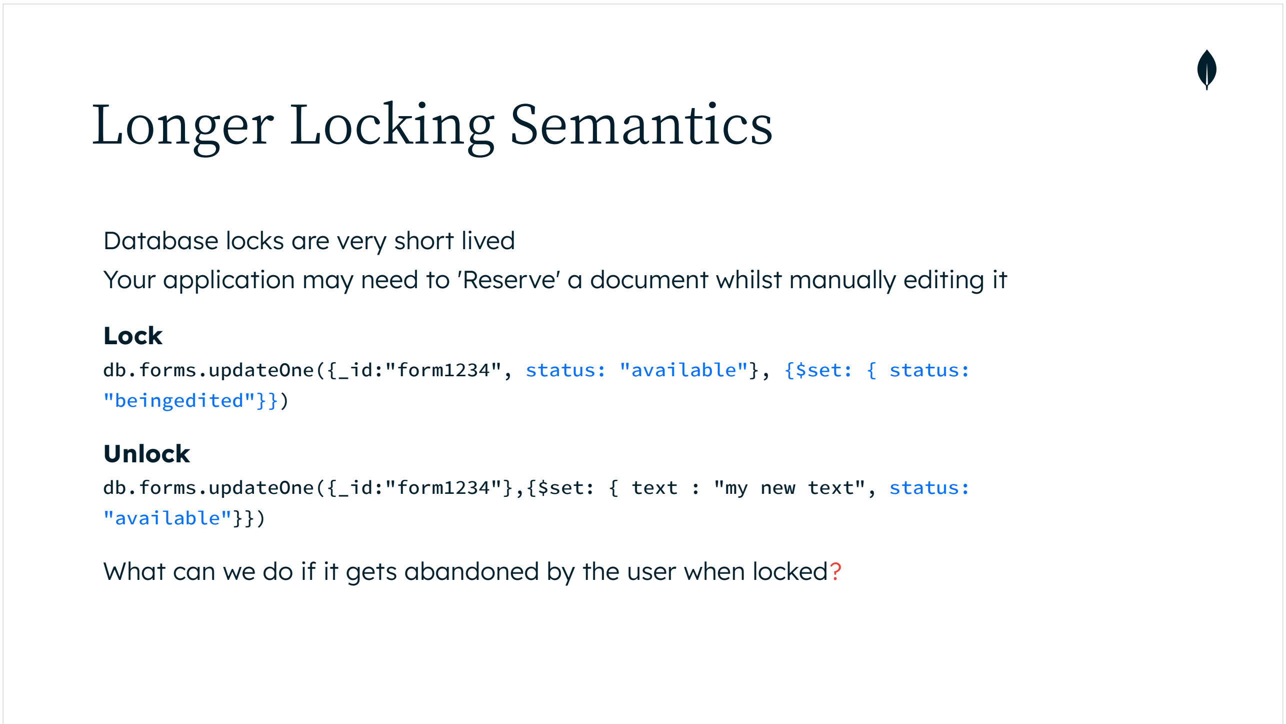
MongoDB has to update the document and possibly indexes

- This must be atomic
- Uses an Internal transaction
- Supports Multi-version concurrency

This fails when committing if there is a contention

- **MongoDB automatically retries until success** - making it **optimistic**
- Extra CPU work if there is a lot of contention

- An update in MongoDB is actually an update of multiple bits of data - the record and its indexes.
- There is an idea of an internal transaction in the database handling this - begin, change files/data, commit
- In MongoDB, if there is contention for a record, this fails in the commit
- If the commit fails, it goes back and retries - so locking is internally optimistic
- This can result in busy work where there is a lot of contention on one record.
- You can see where this contention is the cause of Slow Operations in the database log by looking at the **WriteConflicts** Metric.



Longer Locking Semantics

Database locks are very short lived

Your application may need to 'Reserve' a document whilst manually editing it

Lock

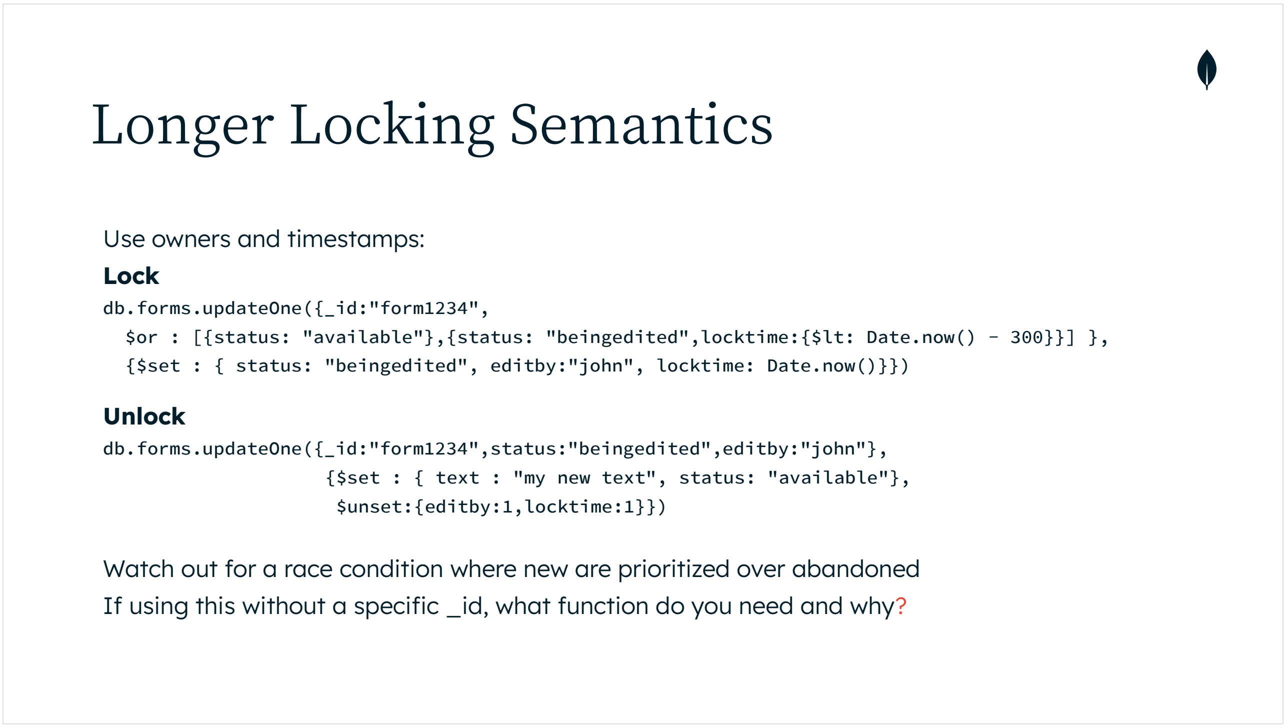
```
db.forms.updateOne({_id:"form1234", status: "available"}, {$set: { status:
"beingedited"}})
```

Unlock

```
db.forms.updateOne({_id:"form1234"},{$set: { text : "my new text", status:
"available"}})
```

What can we do if it gets abandoned by the user when locked?

- Sometimes you want long term locking - for example editing a long piece of text
- For this, you can use application-level locking



Longer Locking Semantics

Use owners and timestamps:

Lock

```
db.forms.updateOne({_id:"form1234",
  $or : [{status: "available"},{status: "beingedited",locktime:{$lt: Date.now() - 300}}] },
  {$set : { status: "beingedited", editby:"john", locktime: Date.now()}})
```

Unlock

```
db.forms.updateOne({_id:"form1234",status:"beingedited",editby:"john"},
                    {$set : { text : "my new text", status: "available"},
                     $unset:{editby:1,locktime:1}})
```

Watch out for a race condition where new are prioritized over abandoned

If using this without a specific _id, what function do you need and why?

- Watch out for race condition - available would come first in the index, so available would be found in preference to being edited but timed out. You might need to use different words or numeric statuses.
- Without a specific _id, you need to know what document you grabbed - for that, you want findOneAndUpdate() - this is atomic too.

Quiz Time!

#1. Which of these statements are true about MongoDB locks?

A

Updates exclusively lock the whole collection for writes

B

Inserts exclusively lock the whole collection for writes

C

Updates on a document block reads for that document

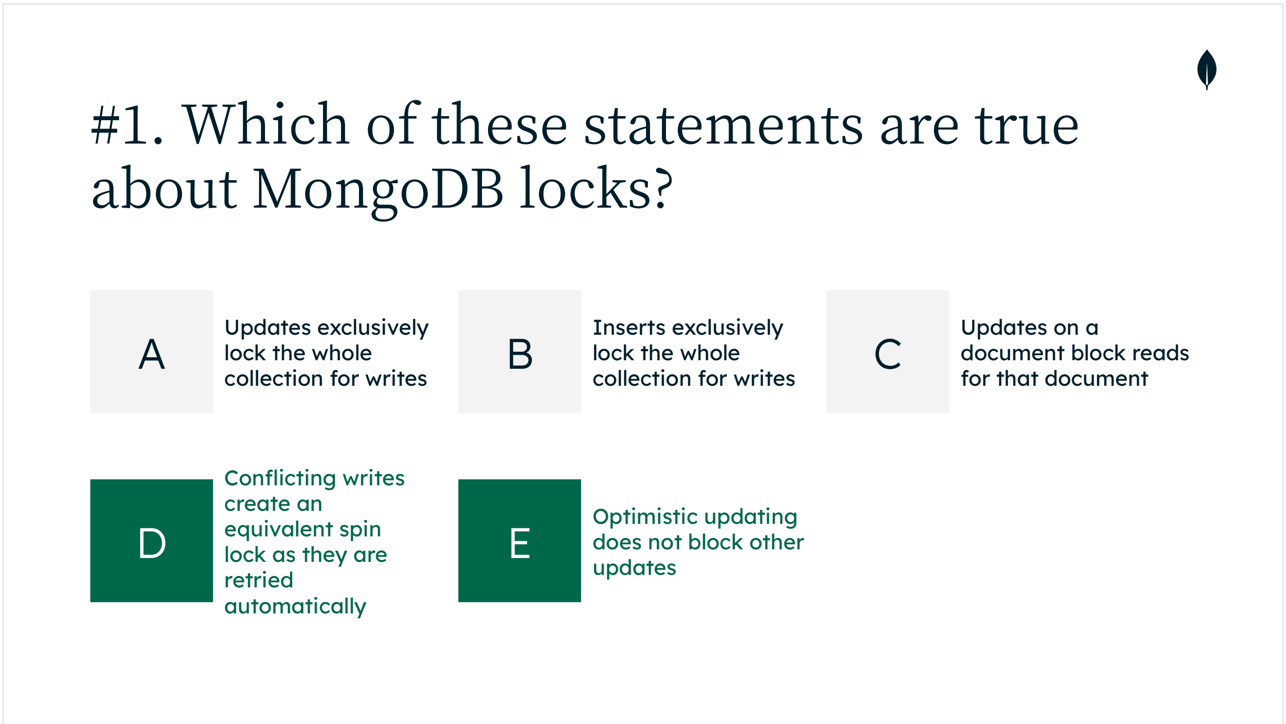
D

Conflicting writes create an equivalent spin lock as they are retried automatically

E

Optimistic updating does not block other updates

Answer in the next slide.



#1. Which of these statements are true about MongoDB locks?

A Updates exclusively lock the whole collection for writes

B Inserts exclusively lock the whole collection for writes

C Updates on a document block reads for that document

D Conflicting writes create an equivalent spin lock as they are retried automatically

E Optimistic updating does not block other updates

Answer: D & E

#2. If a query looks for documents where "field x is less than the number 5", are documents with NULL or is missing x field being returned? Select the best answer.

A Yes, always

B No, never

C Depends on whether you are using .find() or .aggregate()

D Yes for NULL values, but No where the x field is missing

E Depends on the type of the number 5 (double or int) in the query

Answer in the next slide.

#2. If a query looks for documents where "field x is less than the number 5", are documents with NULL or is missing x field being returned? Select the best answer.

A Yes, always

B No, never

[✓ C] Depends on whether you are using .find() or .aggregate()

D Yes for NULL values, but No where the x field is missing

E Depends on the type of the number 5 (double or int) in the query