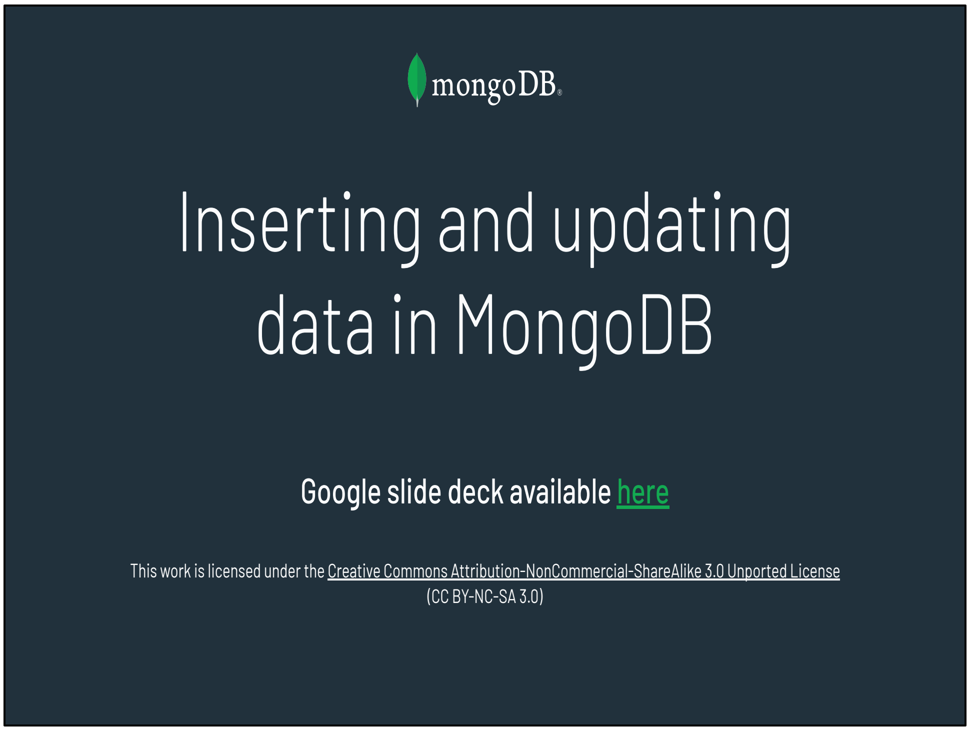
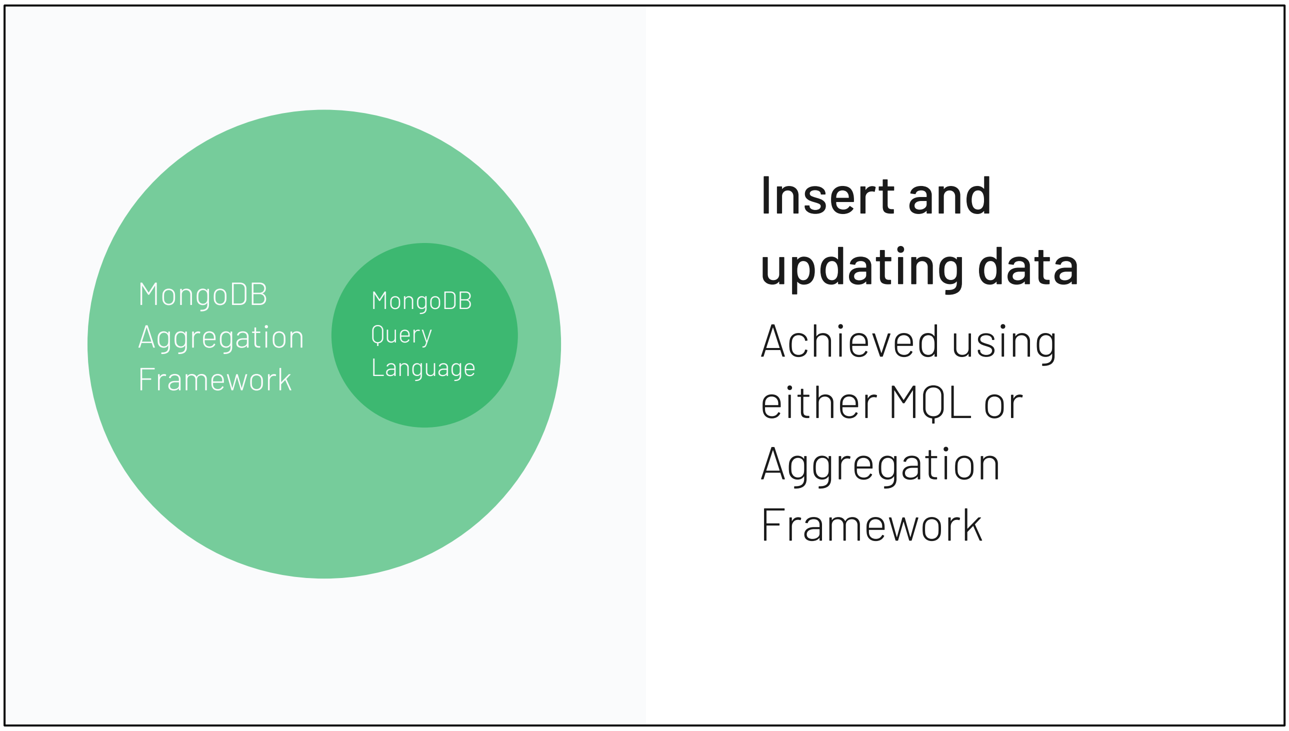


# Inserting and updating data in MongoDB

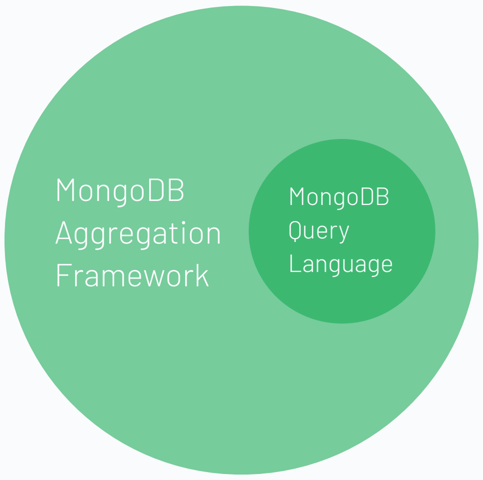
Google slide deck available here

In this lesson, we'll recap briefly on some aspects previously covered on updating and inserting but with some additional context around how best to structure these operations to be performant in terms of the database.

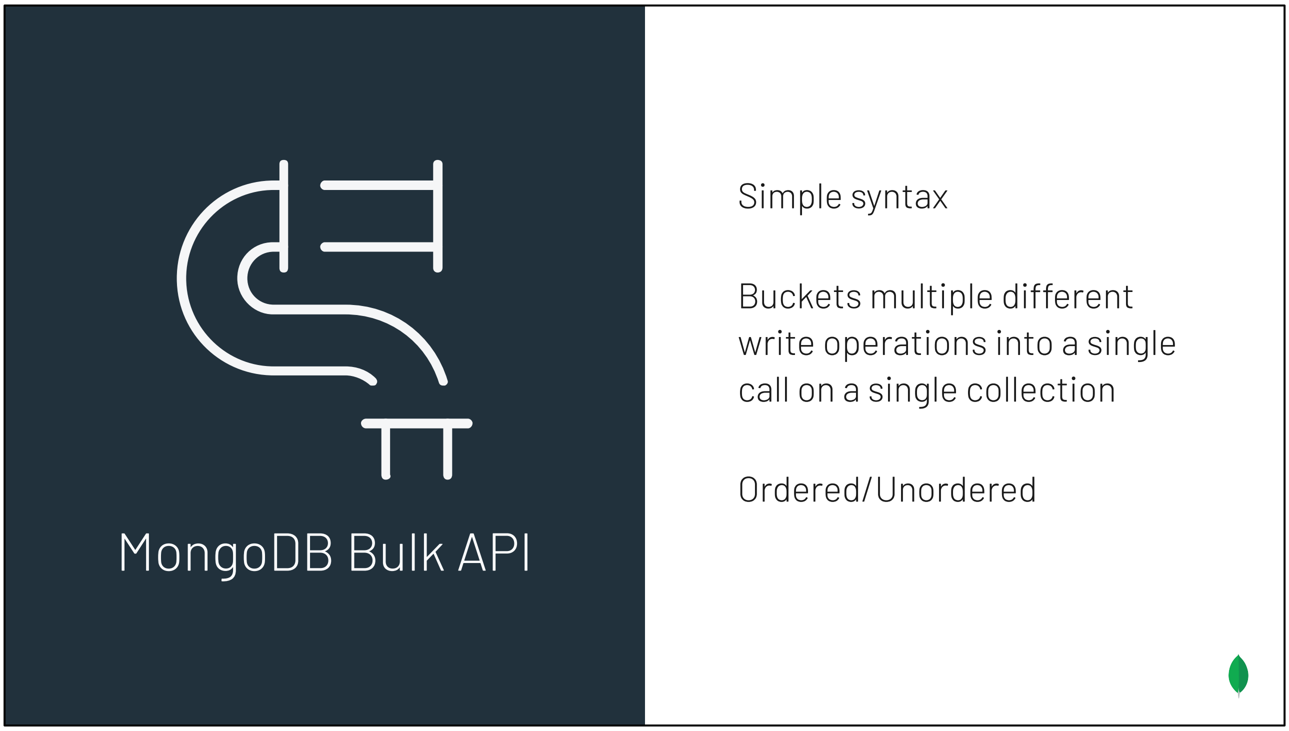


## Insert and updating data

Achieved using either MQL or Aggregation Framework



In previous lessons we have introduced the MongoDB Aggregation Framework and on the MongoDB Query Language (MQL)and we introduced methods of inserting and updating data to a MongoDB database. In this lesson, we'll expand on what we covered previously to introduce the MongoDB Bulk API. The Bulk API is similar to MQL but was designed to allow for situations where large numbers of operations could be performed with a lower overhead than using MQL.



## MongoDB Bulk API

Simple syntax

Buckets multiple different write operations into a single call on a single collection

Ordered/Unordered

In this lesson, we will also introduce the MongoDB Bulk API as another approach to performing multiple different write operations on a single collection using a simple syntax.

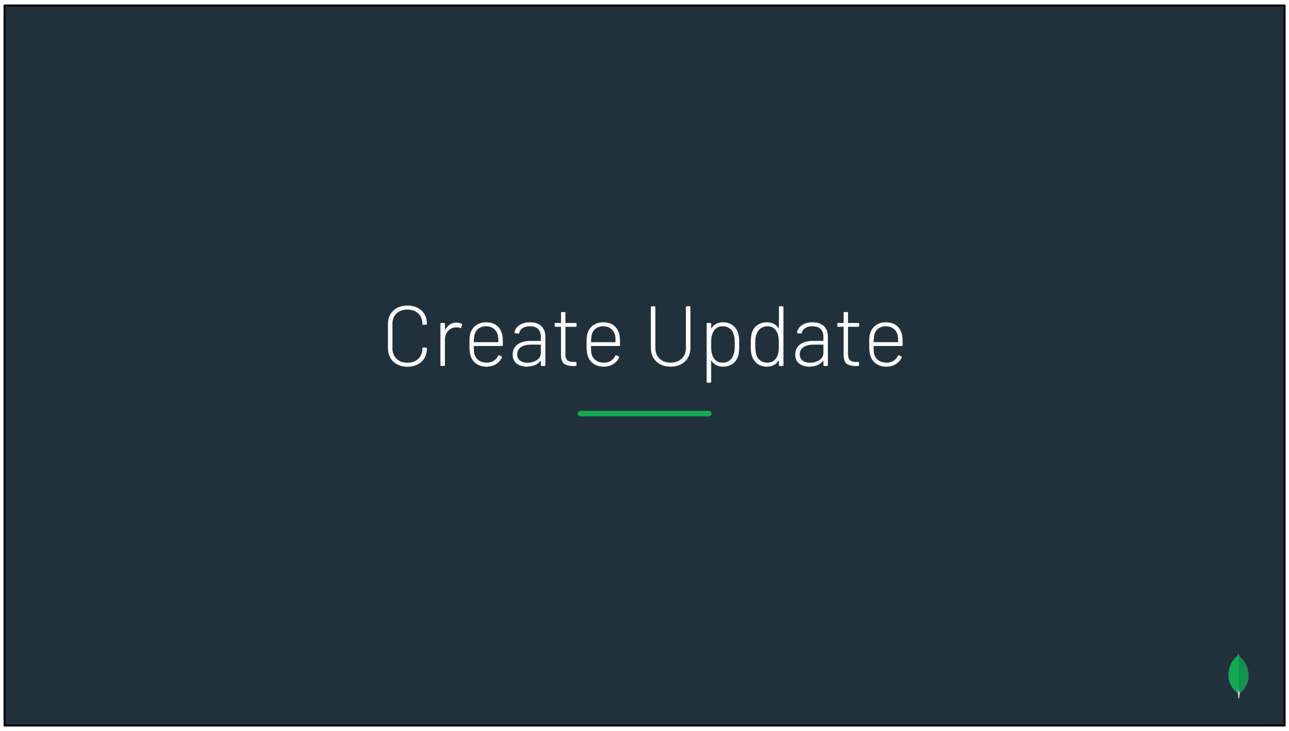
Firstly to reiterate the syntax for the Bulk API is simple, you create the list to hold the operations and you add the operations to the list. The entire list can then be executed against the database.

The list of operations in the Bulk API can be seen as an efficient mechanism to bucket multiple different write operations into a single call on a single collection to the database.

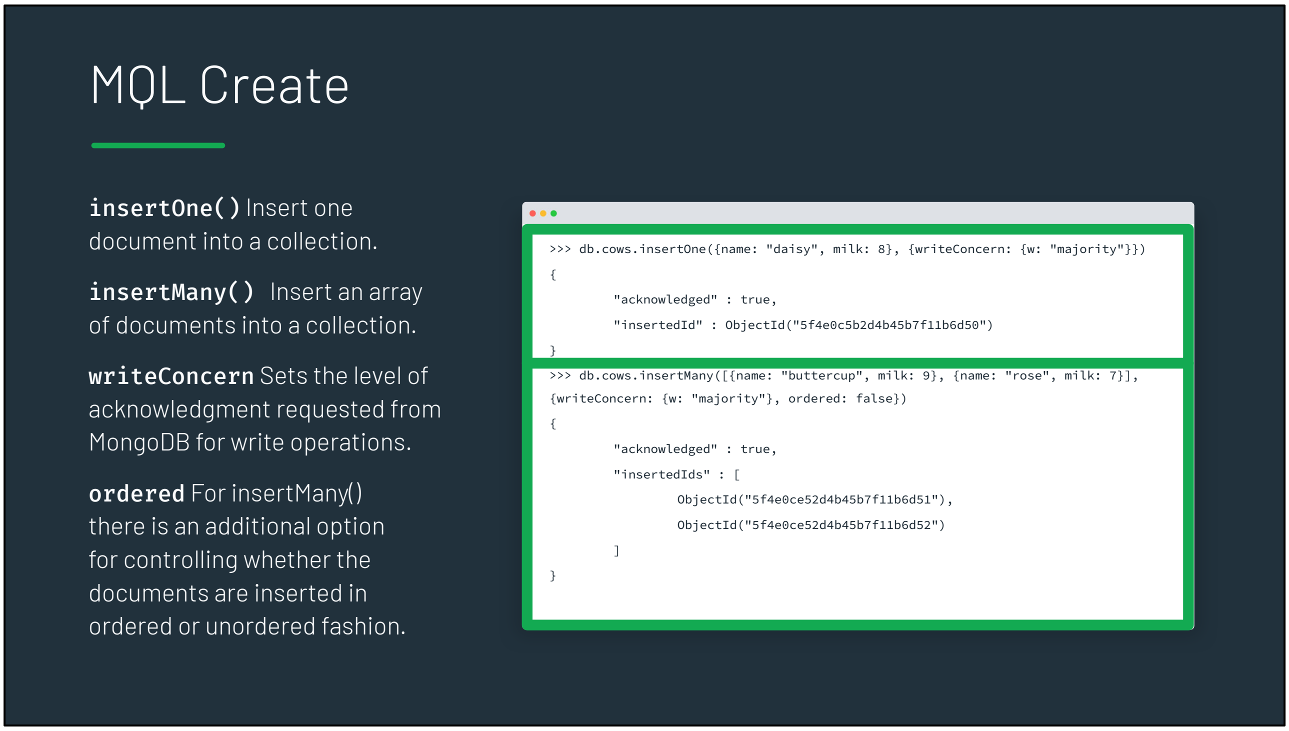
In terms of buckets, you can combine multiple updates, inserts, deletes operations on a single collection within a single call to the database using the Bulk API.

The Bulk API allows for ordered or unordered execution of the operations within the call.

Ordered bulk API operations are performed serially against the collection.
Unordered bulk API operations are performed in parallel against the collection.


## Create Update

Let's firstly look at the Bulk API in terms of create and of update operations.



## MQL Create

**insertOne()** Insert one document into a collection.

**insertMany()** Insert an array of documents into a collection.

**writeConcern** Sets the level of acknowledgment requested from MongoDB for write operations.

**ordered** For insertMany() there is an additional option for controlling whether the documents are inserted in ordered or unordered fashion.

```
>>> db.cows.insertOne({name: "daisy", milk: 8}, {writeConcern: {w: "majority"}})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5f4e0c5b2d4b45b7f11b6d50")
}
```

```
>>> db.cows.insertMany([{name: "buttercup", milk: 9}, {name: "rose", milk: 7}],
{writeConcern: {w: "majority"}, ordered: false})
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5f4e0ce52d4b45b7f11b6d51"),
                ObjectId("5f4e0ce52d4b45b7f11b6d52")
        ]
}
```

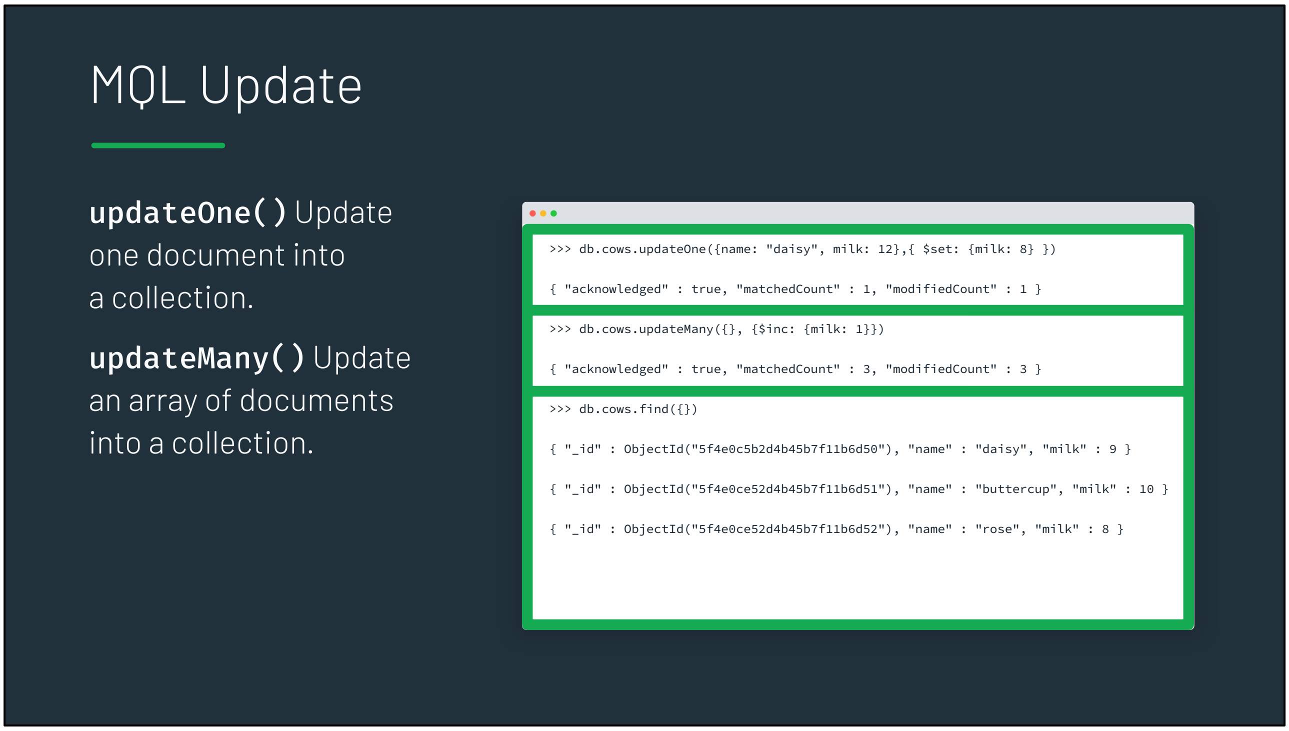
In this section, we'll explore a little more around the Create and Update (or the C and U in CRUD) functions in MQL.

We'll start with the Create functions of insertOne() and insertMany().
insertOne() inserts one document into a collection.
The main difference is insertMany() takes an array of documents whilst insertOne() only takes a single document.

Both functions can take an optional writeConcern parameter. The writeConcern sets the level of acknowledgement requested from the MongoDB database for write operations.

The insertMany() function can also optionally take an ordered parameter. The ordered parameter determines if the documents must be inserted in the order they are present in the array, the default is to insert documents ordered in the way they are present in the array.

The code example on the slide is an example of firstly using insertOne and then using insertMany.



## MQL Update

**`updateOne()`** Update one document into a collection.

**`updateMany()`** Update an array of documents into a collection.

```
>>> db.cows.updateOne({name: "daisy", milk: 12},{ $set: {milk: 8} })

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

```
>>> db.cows.updateMany({}, {$inc: {milk: 1}})

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

```
>>> db.cows.find({})

{ "_id" : ObjectId("5f4e0c5b2d4b45b7f11b6d50"), "name" : "daisy", "milk" : 9 }

{ "_id" : ObjectId("5f4e0ce52d4b45b7f11b6d51"), "name" : "buttercup", "milk" : 10 }

{ "_id" : ObjectId("5f4e0ce52d4b45b7f11b6d52"), "name" : "rose", "milk" : 8 }
```

Update in similar fashion to Create has two functions, updateOne and updateMany, which operate on a single document and on an array of documents respectively.

updateOne will update a single document into a collection.
updateMany will update an array of documents into a collection.

Both functions take a filter document which specifies which documents to be updated, if the filter document is empty the first document in the collection is updated. These functions also take a second document as an argument which contains the various options that can be configured.

The default behaviour is not to insert a new document if the update cannot find a suitable document or documents that match the filter document. It is possible using the upsert: True parameter in the second document supplied to the function (the options document) to set that if a document isn't found that a new one will be inserted (hence the term upsert - update and insert).

In the code example on screen/on slide, we build on the example used in the MQL Create slide.
Let's revert the change we made in the MQL Read slide where we set the milk to 12 from 8 for the cow called daisy. We can see in the output of the updateOne, that it both found and updated 1 document.

Let's update all of the documents in the cows collection and increment/add their milk fields by 1.

Let’s then use a find to look at the result for a subset of the documents, we can see that for cow named daisy, we’d set the milk field to 8 and then this was incremented by 1 so it’s now showing 9.

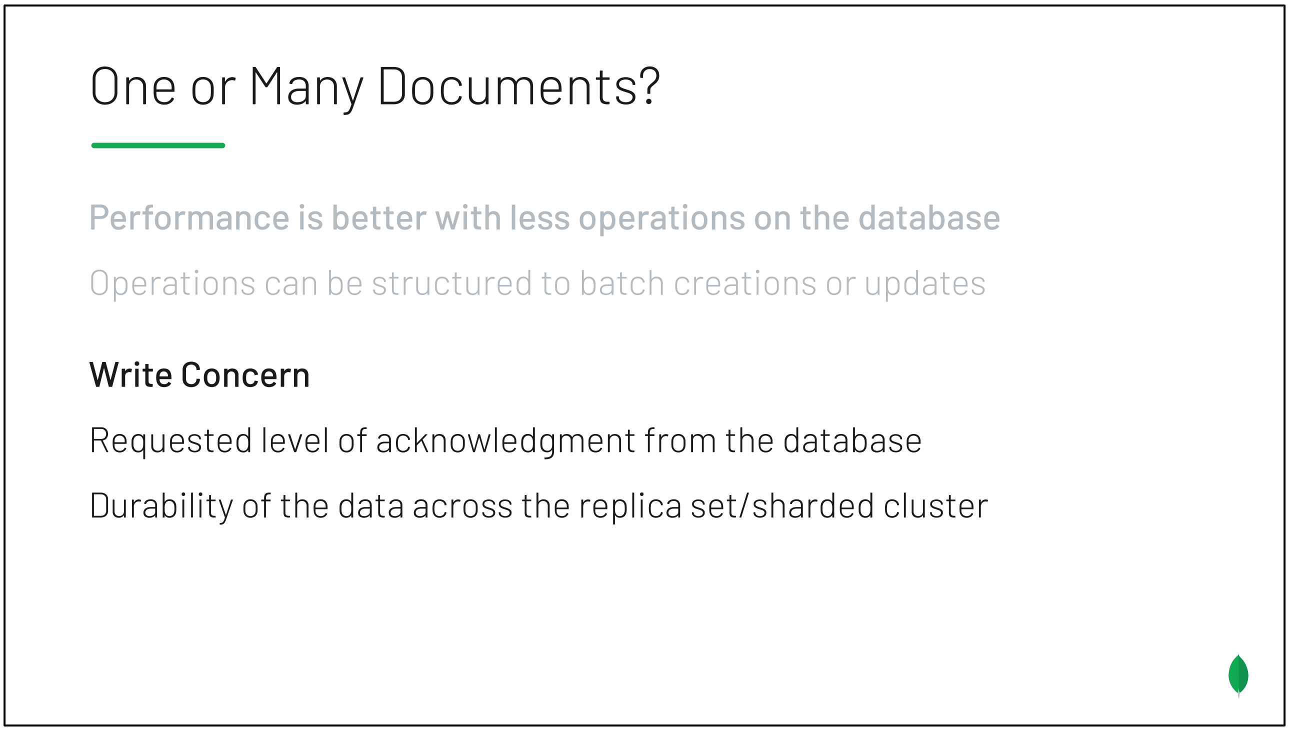
## One or Many Documents?

**Performance is better with less operations on the database**

Operations can be structured to batch creations or updates

In terms of the performance with regards a database, it is more performant to limit the number of operations against it by making fewer if larger operations with the xxxMany() create / update functions.

These operations can ideally be structured as batch creations or updates.



## One or Many Documents?

**Performance is better with less operations on the database**

Operations can be structured to batch creations or updates

**Write Concern**

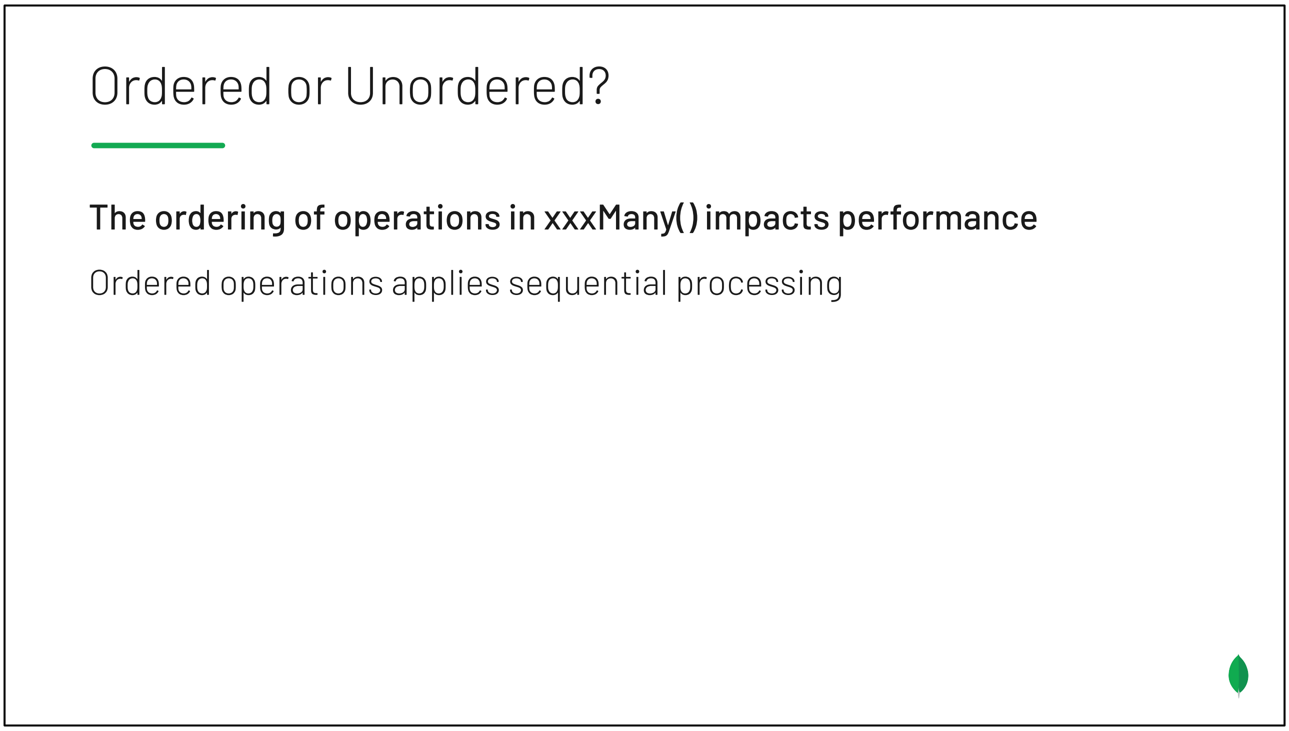
Requested level of acknowledgment from the database

Durability of the data across the replica set/sharded cluster

We have mentioned Write Concern in others lessons but we haven't spoken about it in terms of performance. It controls the level of acknowledgement in terms of the write operation requested from the database.

Specifically, this is configurable and is the desired number of acknowledges from other members in the replica set that they have received and applied that specific write operation. After the desired number of members acknowledge the write operation, the operation is marked as successful and passed back.

The write concern essentially sets the durability of the data by selecting a higher number of nodes than one then once the operation is successful you know that at least one and potentially more members have copies of the data so that failures of any one node will not lose that data. All write operations will eventually be replicated to all members but this aspect is concerned with the window of time from the operation but before the data has been durable written across all members of the replica set. It seeks to minimise the dangers of data loss in that period.



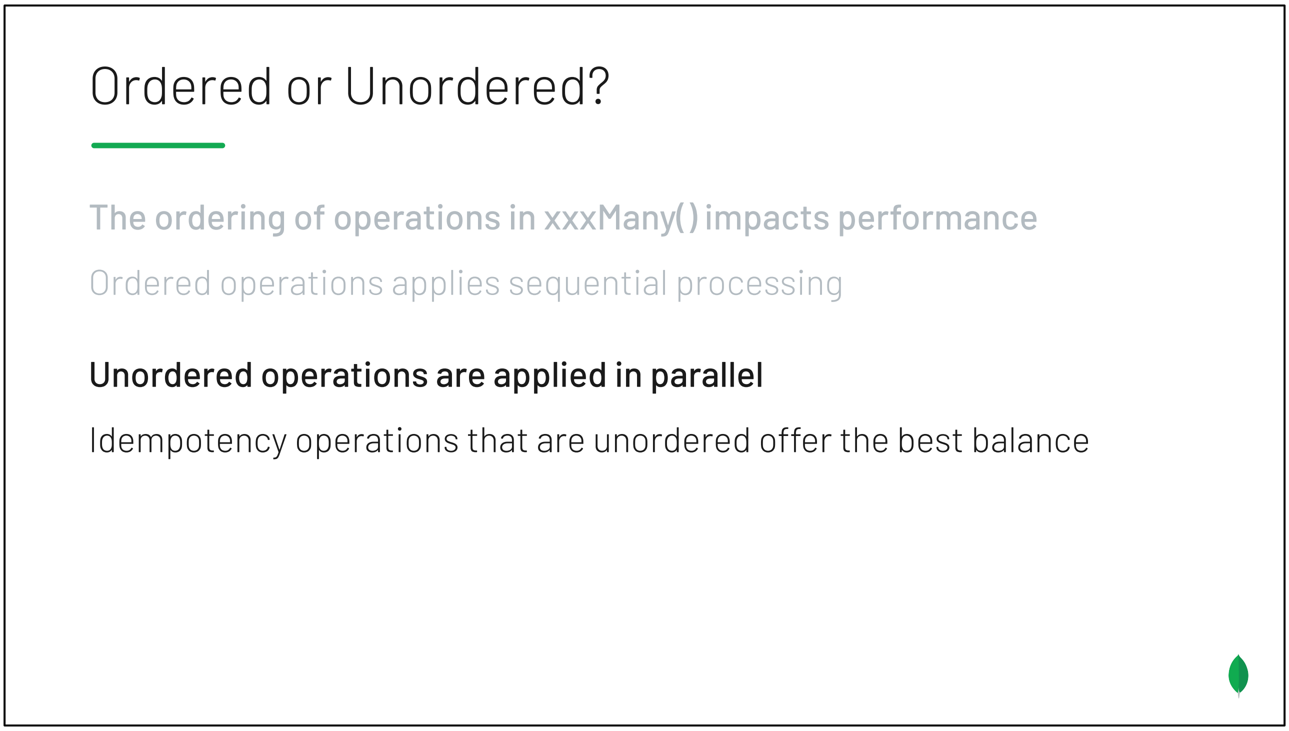
## Ordered or Unordered?

**The ordering of operations in xxxMany() impacts performance**

Ordered operations applies sequential processing

Ordering of the insertion and update operations in xxxMany() functions can have a performance impact due to how they are processed.

Ordered operations are applied in a sequential order, so A-B-C with each operation occurring after the prior has finished.



## Ordered or Unordered?

The ordering of operations in xxxMany() impacts performance

Ordered operations applies sequential processing

**Unordered operations are applied in parallel**

Idempotency operations that are unordered offer the best balance

Unordered operations are applied in parallel so A & B & C can be write to the database at the same time. This may have undesired impacts as the operations may also be reordered so it is C & B & A so if data may be changed for certain updates in a sequence different to that expected.

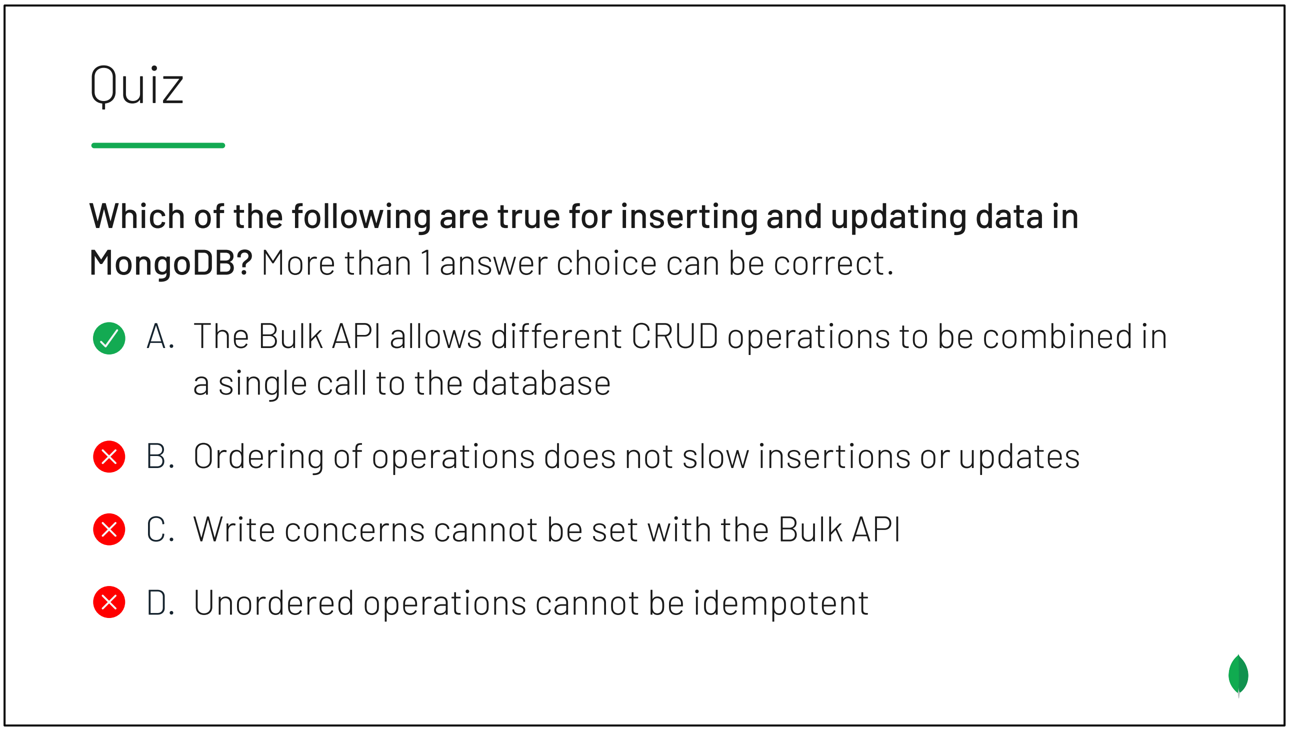
Idempotent operations can be applied many times without changing the final result after they have been applied the first time. These operations will produce the same results whether applied once or many time.

## Quiz

## Quiz

**Which of the following are true for inserting and updating data in MongoDB?** More than 1 answer choice can be correct.

- A. The Bulk API allows different CRUD operations to be combined in a single call to the database
- B. Ordering of operations does not slow insertions or updates
- C. Write concerns cannot be set with the Bulk API
- D. Unordered operations cannot be idempotent



## Quiz

**Which of the following are true for inserting and updating data in MongoDB?** More than 1 answer choice can be correct.

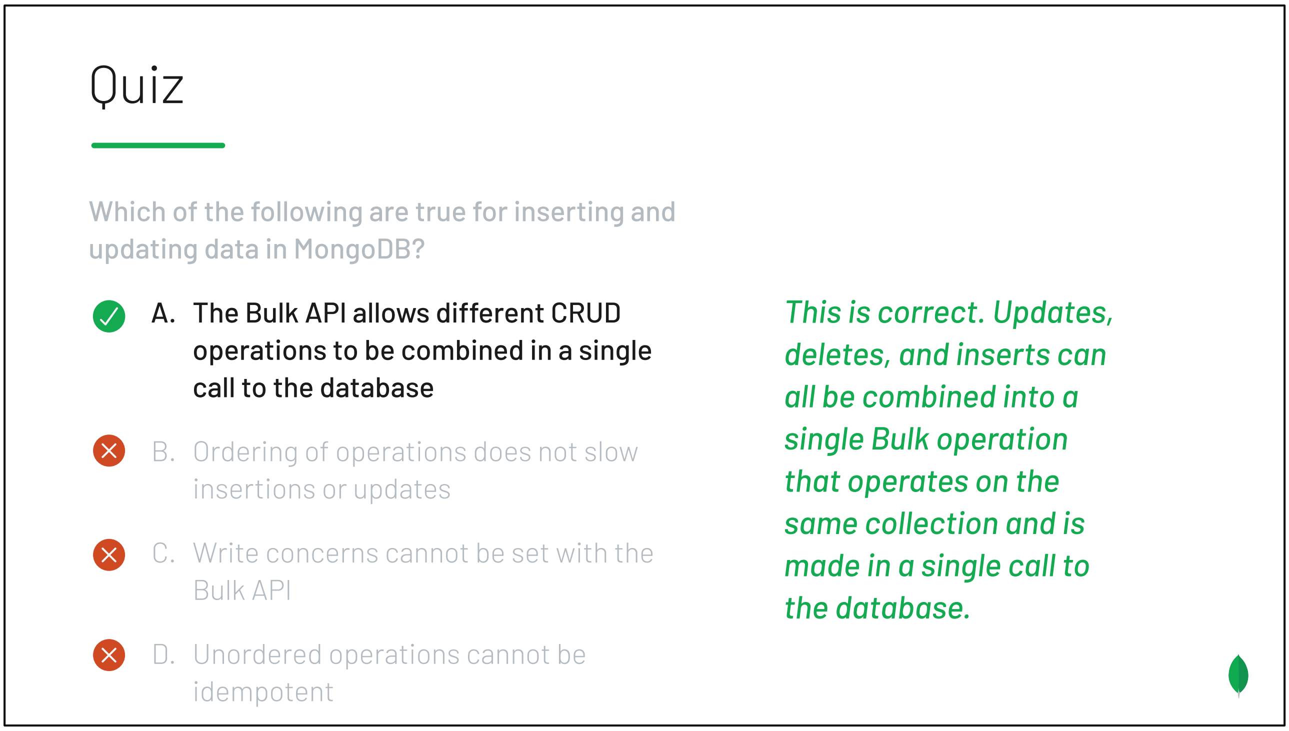
✓ A. The Bulk API allows different CRUD operations to be combined in a single call to the database

✗ B. Ordering of operations does not slow insertions or updates

✗ C. Write concerns cannot be set with the Bulk API

✗ D. Unordered operations cannot be idempotent

CORRECT: The Bulk API allows different CRUD operations to be combined in a single call to the database - This is correct, updates, deletes, and inserts can all be combined into a single Bulk operation that operates on the same collection and is made in a single call to the database.
INCORRECT: Ordering of operations does not slow insertions or updates - The ordering does slow down operations when compared to unordered operations. Ordered operations are processed sequentially whilst unordered are processed in parallel, hence these can be faster to the inherent parallelisation.
INCORRECT: Write concerns cannot be set with the Bulk API - write concerns can be set with the Bulk API.
INCORRECT: Unordered operations cannot be idempotent - any operation whether ordered or unordered can be idempotent, not every operation is idempotent but the ordering does not impact this.



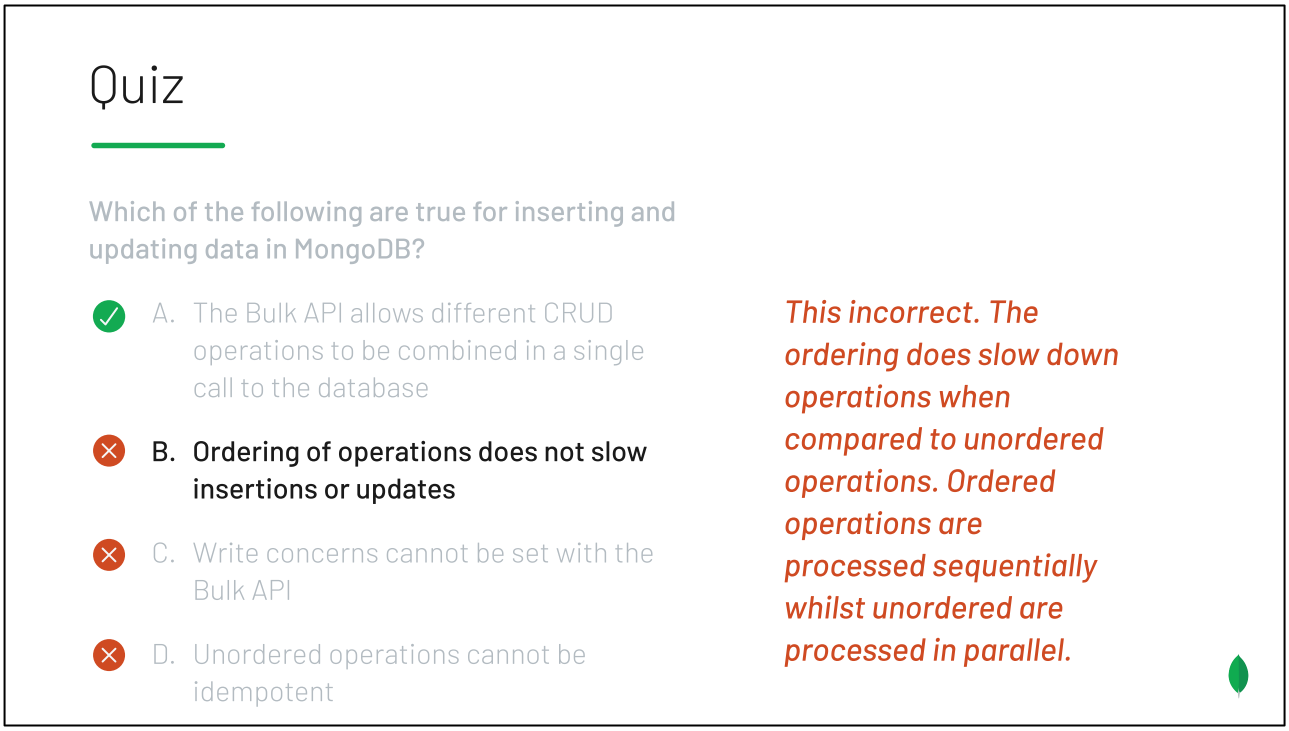
## Quiz

Which of the following are true for inserting and updating data in MongoDB?

- ✓ A. The Bulk API allows different CRUD operations to be combined in a single call to the database
- ✗ B. Ordering of operations does not slow insertions or updates
- ✗ C. Write concerns cannot be set with the Bulk API
- ✗ D. Unordered operations cannot be idempotent

*This is correct. Updates, deletes, and inserts can all be combined into a single Bulk operation that operates on the same collection and is made in a single call to the database.*

CORRECT: The Bulk API allows different CRUD operations to be combined in a single call to the database - This is correct. Updates, deletes, and inserts can all be combined into a single Bulk operation that operates on the same collection and is made in a single call to the database.



## Quiz

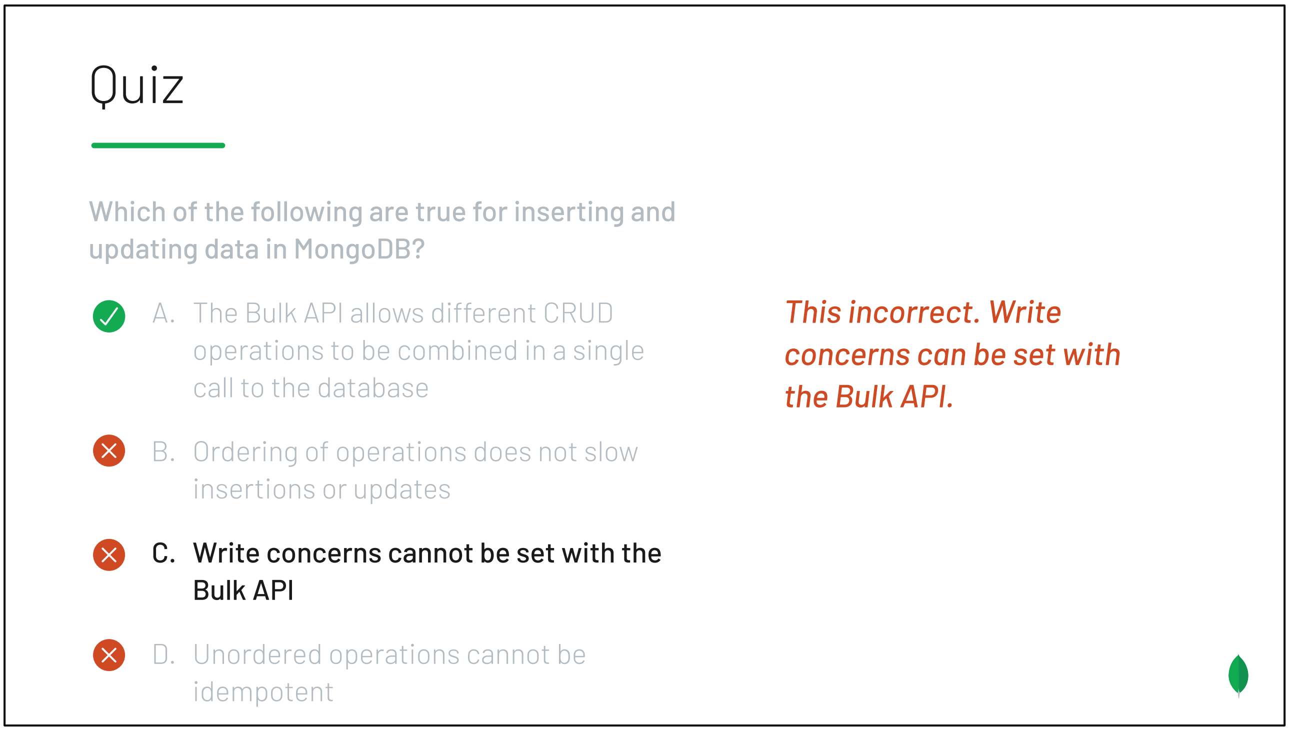
Which of the following are true for inserting and updating data in MongoDB?

- ✓ A. The Bulk API allows different CRUD operations to be combined in a single call to the database
- ✗ B. Ordering of operations does not slow insertions or updates
- ✗ C. Write concerns cannot be set with the Bulk API
- ✗ D. Unordered operations cannot be idempotent

*This incorrect. The ordering does slow down operations when compared to unordered operations. Ordered operations are processed sequentially whilst unordered are processed in parallel.*

INCORRECT: Ordering of operations does not slow insertions or updates - This is incorrect. The ordering does slow down operations when compared to unordered operations. Ordered operations are processed sequentially whilst unordered are processed in parallel.

Note: Parallel processed operations are faster due to the inherent parallelisation.



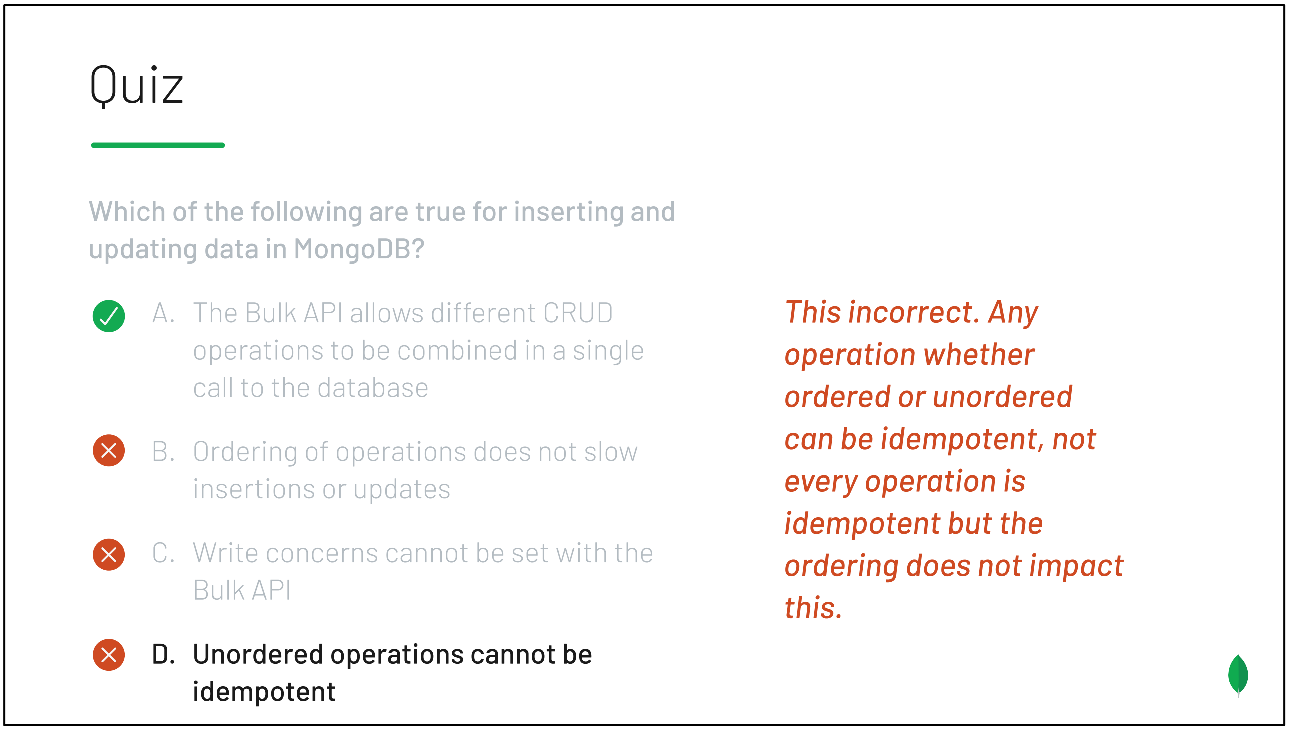
## Quiz

Which of the following are true for inserting and updating data in MongoDB?

- ✓ A. The Bulk API allows different CRUD operations to be combined in a single call to the database
- ✗ B. Ordering of operations does not slow insertions or updates
- ✗ **C. Write concerns cannot be set with the Bulk API**
- ✗ D. Unordered operations cannot be idempotent

*This incorrect. Write concerns can be set with the Bulk API.*

INCORRECT: Write concerns cannot be set with the Bulk API - This is incorrect. Write concerns can be set with the Bulk API.



## Quiz

Which of the following are true for inserting and updating data in MongoDB?

- ✓ A. The Bulk API allows different CRUD operations to be combined in a single call to the database
- ✗ B. Ordering of operations does not slow insertions or updates
- ✗ C. Write concerns cannot be set with the Bulk API
- ✗ **D. Unordered operations cannot be idempotent**

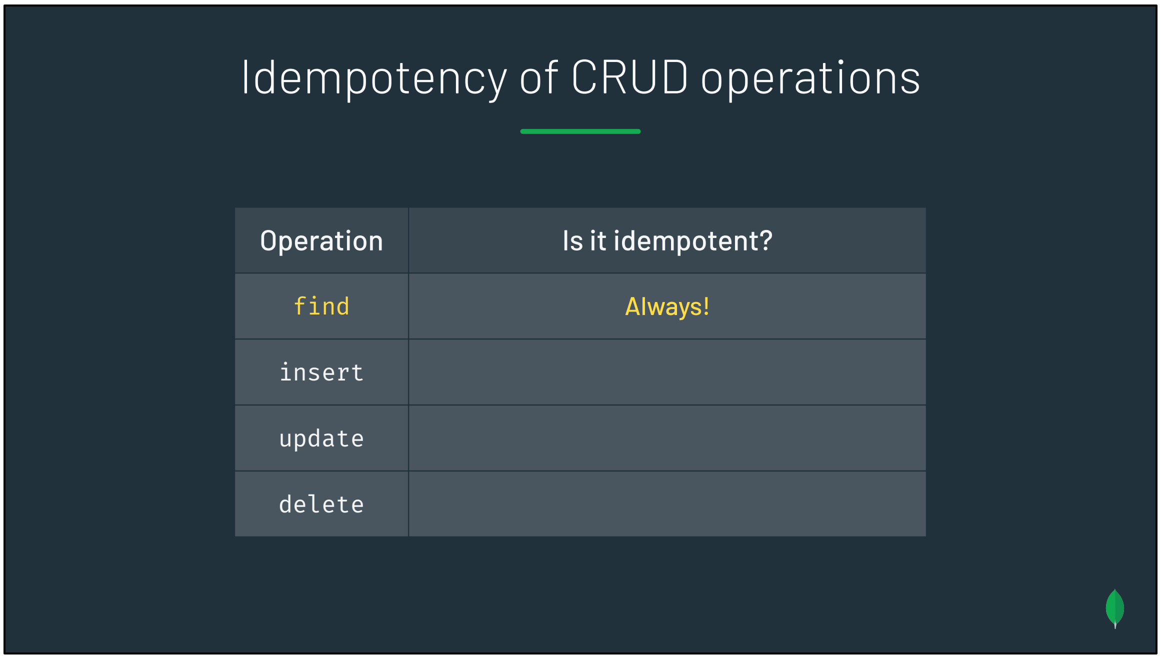
*This incorrect. Any operation whether ordered or unordered can be idempotent, not every operation is idempotent but the ordering does not impact this.*

INCORRECT: Unordered operations cannot be idempotent - This is incorrect. Any operation whether ordered or unordered can be idempotent, not every operation is idempotent but the ordering does not impact this.

## Idempotency

Idempotency is the term used to describe where a given operation will result in the same output when given the same input, whether it is run once or run many times.

Let's look at idempotency and the CRUD operations in MongoDB to see how to view the various type of operations in the database from a idempotent viewpoint.

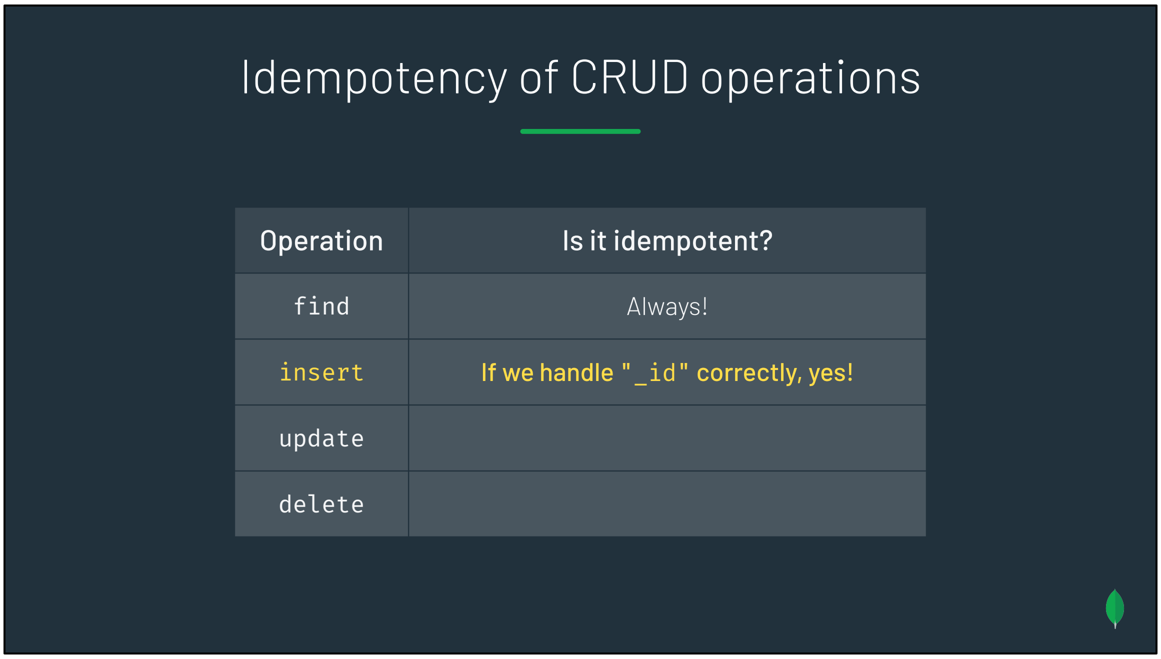


## Idempotency of CRUD operations

| Operation | Is it idempotent? |
|---|---|
| `find` | Always! |
| `insert` | |
| `update` | |
| `delete` | |

Let's start by looking at the CRUD operations with find().

A find operation will be idempotent as it returns the data at that point without changing it. Whether you find a document once or several times the document returned will be the same (assumption: we're working on the assumption for this example the document hasn't been changed by another process during this period).

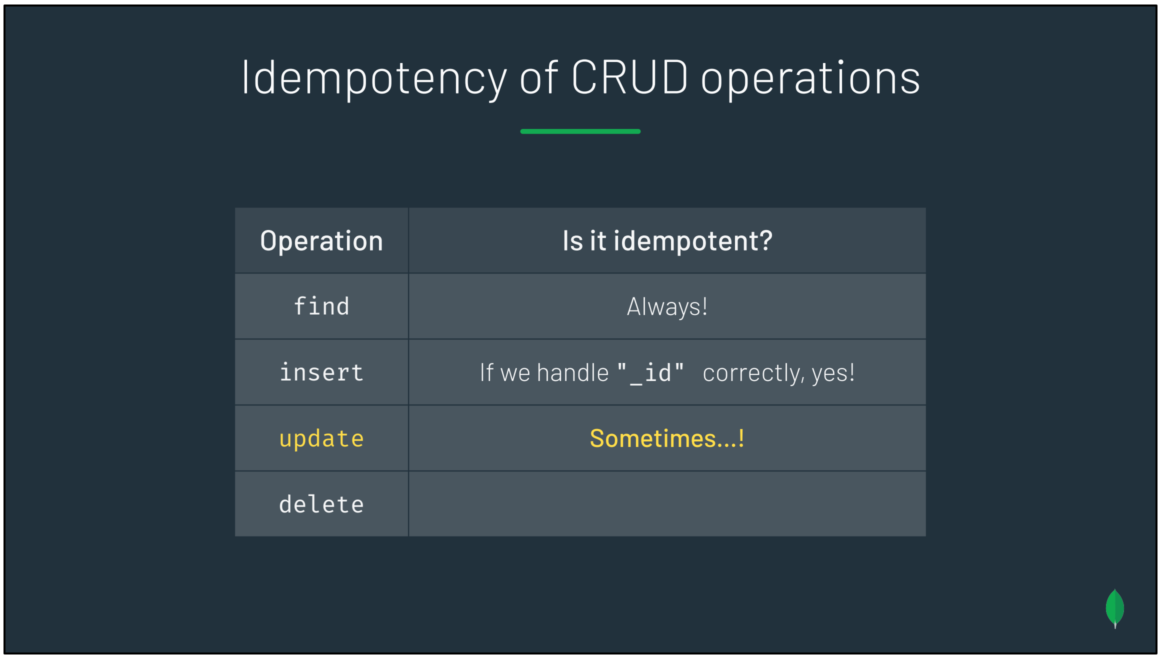


## Idempotency of CRUD operations

| Operation | Is it idempotent? |
|---|---|
| `find` | Always! |
| `insert` | If we handle "`_id`" correctly, yes! |
| `update` | |
| `delete` | |

Let's look at insertion operations next.

Insertion operations are typically idempotent is the _id is used to ensure the uniqueness of the document. MongoDB provides the ObjectID that is typically used for the _id but any unique value is also a candidate. If an insert occurs and the operation was then done again, it will trigger a duplicate key error which allows us to handle the “_id” in code and ignore the error as essential the document exists in the collection already. There may be other errors which we would need to handle but for the case of idempotency we’ll limit to the focus to this aspect.



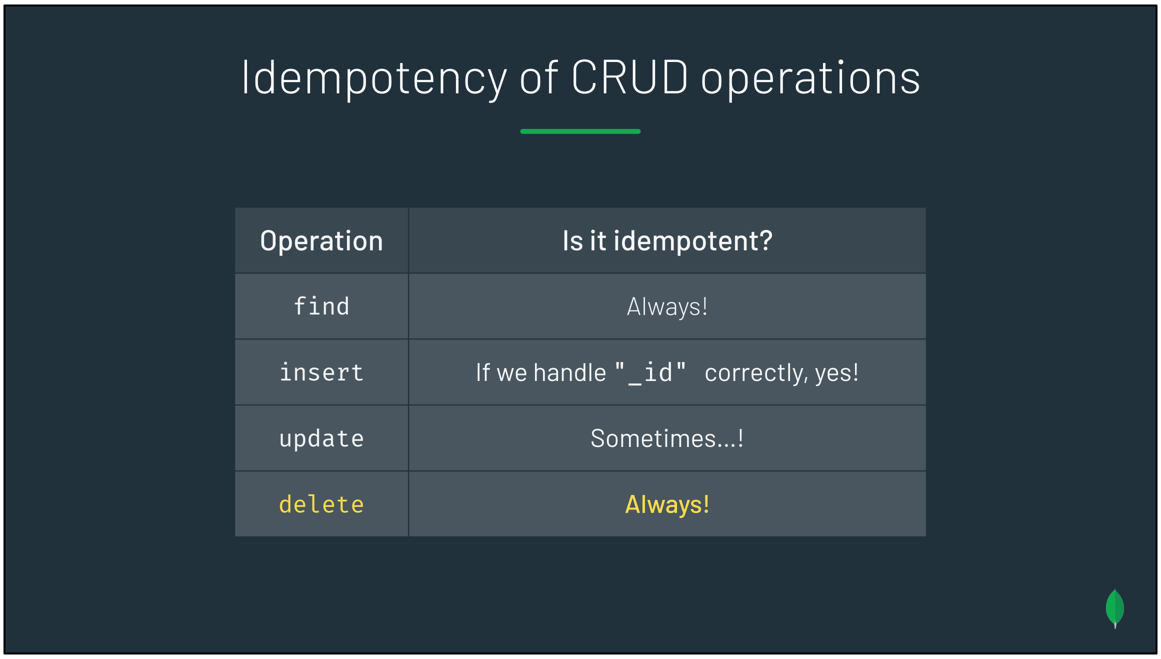
## Idempotency of CRUD operations

| Operation | Is it idempotent? |
|---|---|
| `find` | Always! |
| `insert` | If we handle `"_id"` correctly, yes! |
| `update` | Sometimes...! |
| `delete` | |

Let's now focus on update operations.

Update operations may be problematic when considered in terms of idempotency. This is because some update operators are idempotent whilst others are not. A good example of these is that the $set operator is idempotent as setting a value repeatedly to the same value is idempotent. However, considering the $inc operator which increments the value on each call/operation so each time it is called the value is increased which is clearly not idempotent.

It is possible to break an $inc operation into two operations where firstly a pending or similar token is added to the document and in the second operation both the token and the $inc operation are updated. This is possible in MongoDB as all individual update operations are atomic.

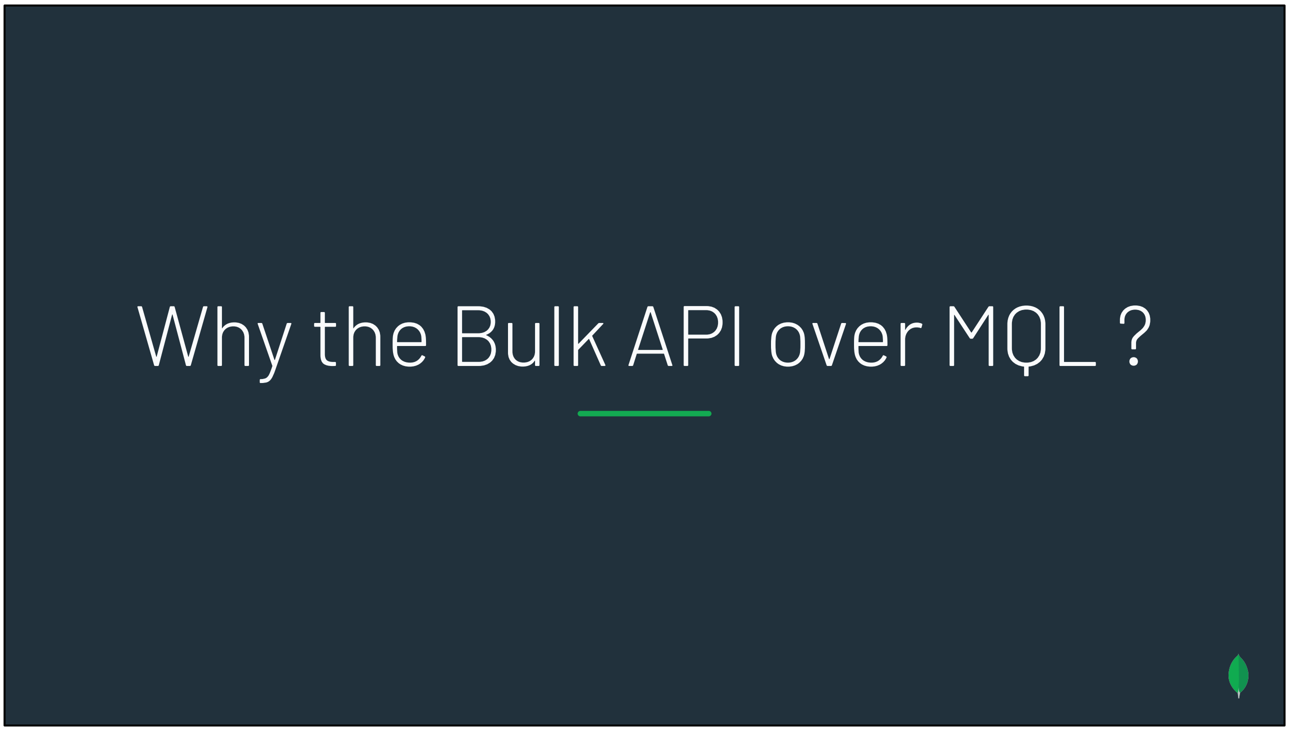


## Idempotency of CRUD operations

| Operation | Is it idempotent? |
|---|---|
| `find` | Always! |
| `insert` | If we handle `"_id"` correctly, yes! |
| `update` | Sometimes...! |
| `delete` | Always! |

Let's look at deletion operations as the final set of CRUD operations in terms of idempotency.

A delete operation will always be idempotent as you can always delete a document and you can repeatedly delete it but once the first operation is applied (it gets deleted) then it won't have any further impact on the state of the data in the collection.



## Why the Bulk API over MQL ?

In the next slides, we'll try to answer the question "Why the Bulk API over MQL?".
This will help understand how and when the Bulk API should be used.

## Why use the Bulk API over xxxMany?

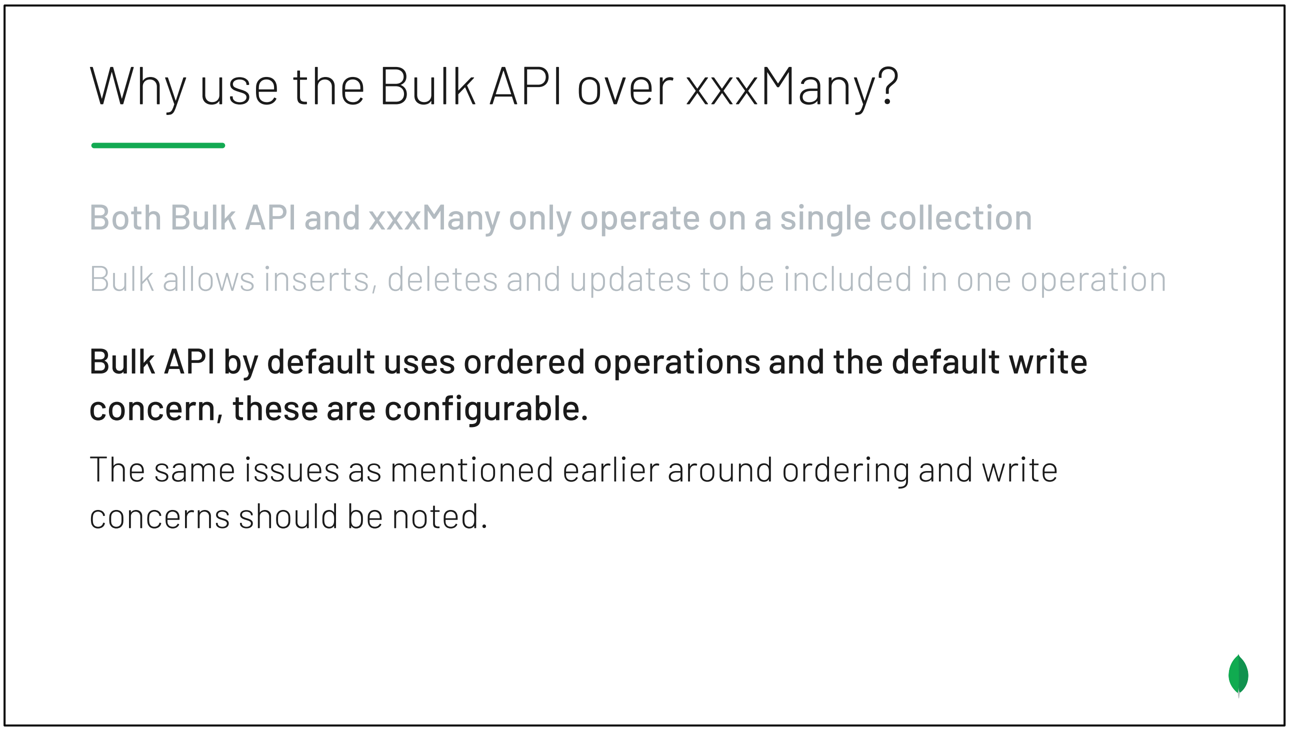
**Both Bulk API and xxxMany only operate on a single collection**

Bulk allows inserts, deletes and updates to be included in one operation

Both the Bulk API and the xxxMany() functions only operate on a single collection. As the Bulk API is then like the other MQL CRUD functions which only work against a single collection, the question arises why would you use the Bulk API over the equvialent xxxMany functions?

The Bulk API allows you to include several different types of CRUD functions (inserts, deletes, and updates) in a single Bulk operation and apply these against the database.

This allows you to combine many different types of write operations into a single call, which will limited to a single collection is already reducing the processing impact as each of the different types would require a single MQL call for just that type.



## Why use the Bulk API over xxxMany?

**Both Bulk API and xxxMany only operate on a single collection**

Bulk allows inserts, deletes and updates to be included in one operation

**Bulk API by default uses ordered operations and the default write concern, these are configurable.**

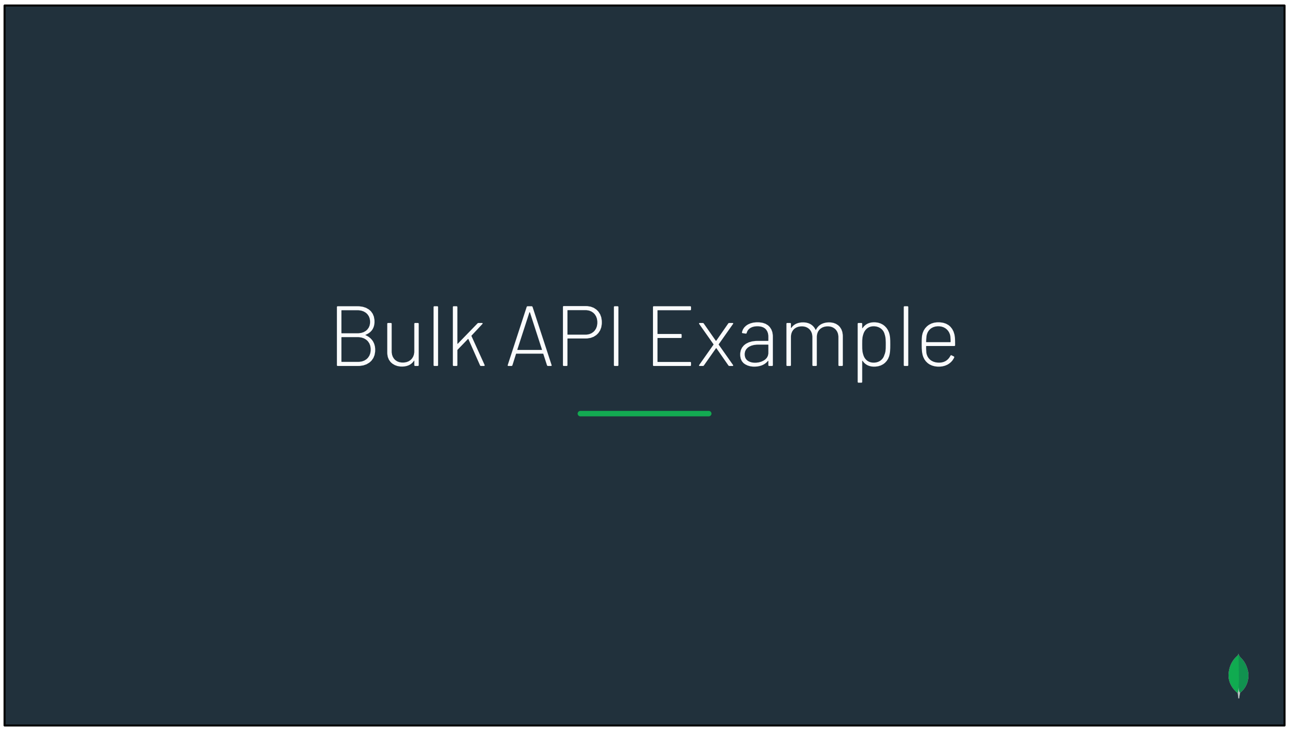
The same issues as mentioned earlier around ordering and write concerns should be noted.

The Bulk API defaults to using ordered operations and the default write concern, both of these are configurable.

The same issues as mentioned earlier around ordering and write concerns should be noted.

For example and as noted earlier in the lesson, you can use unordered operations but you may need to consider that data may be changed out of sequence so if this is an issue then considering ordered operation or idempotency or both may be required.

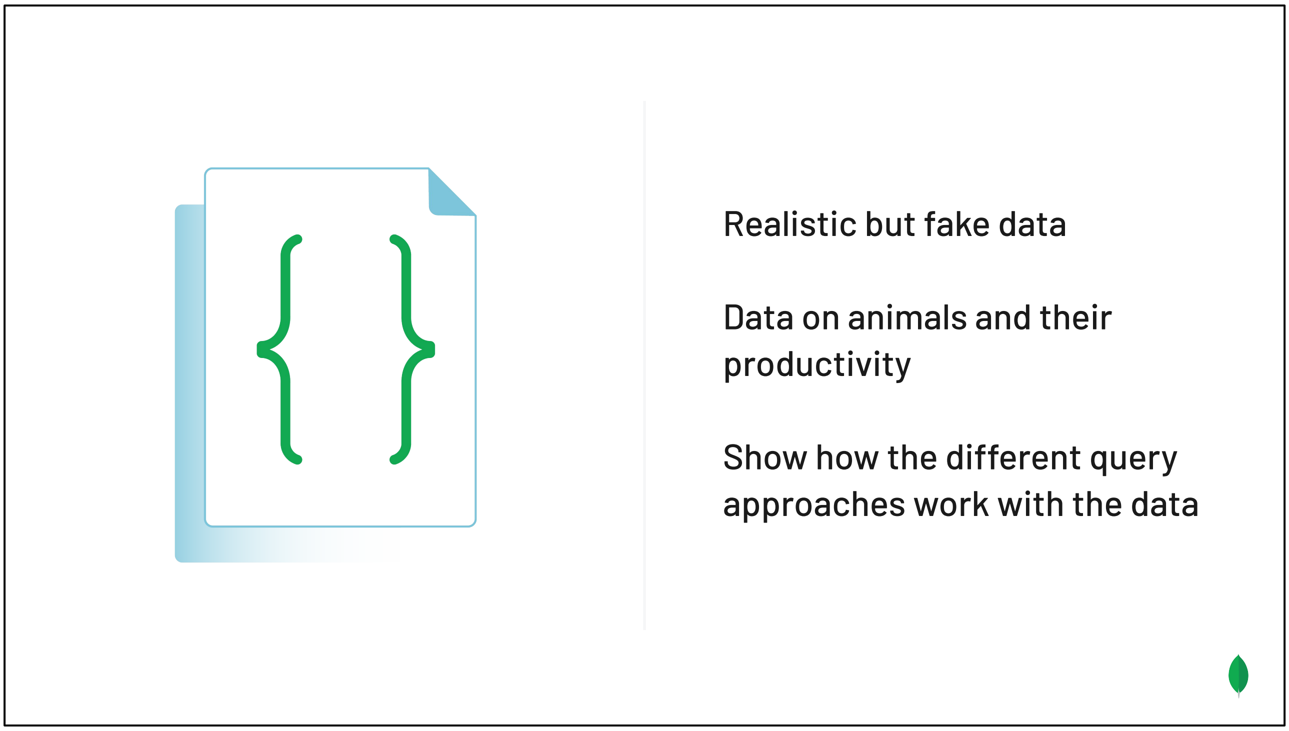
Additionally, the same issues around write concerns and the durability of the data across many nodes as a bottleneck to receiving the successful acknowledgement of the write should also be considered if configuring a different write concern.



## Bulk API Example

Let’s look at using the Bulk API to perform multiple CRUD operations in one call to the database

Let’s look at the Bulk API with the MongoDB Web Shell, visit https://mws.mongodb.com/?version=4.4 in your browser.



Realistic but fake data

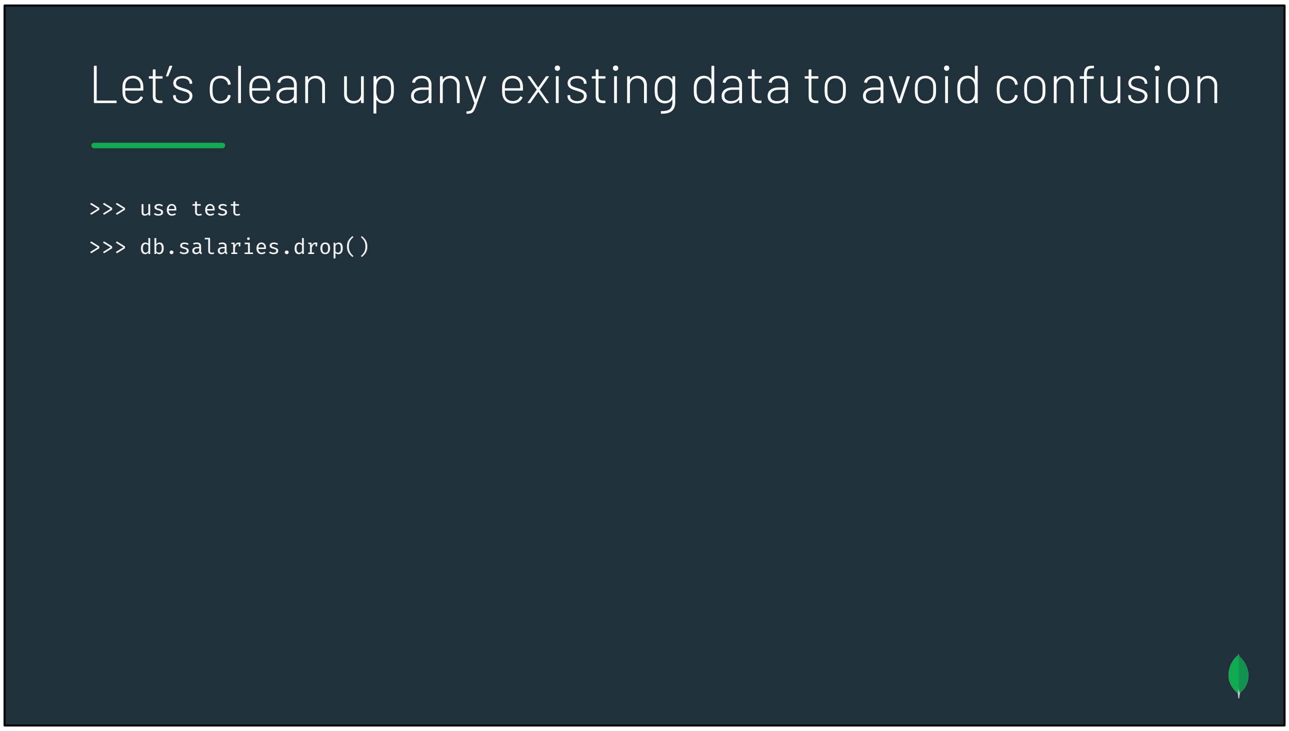
Data on animals and their productivity

Show how the different query approaches work with the data

Firstly, we’re going to create some realistic but fake data to explore using the Bulk API in MongoDB.

This data includes details on the role, the department, the salary for a given employee, in this example we will have one document per fiscal year. .

We'll use this data to highlight some examples of how you can use the Bulk API.



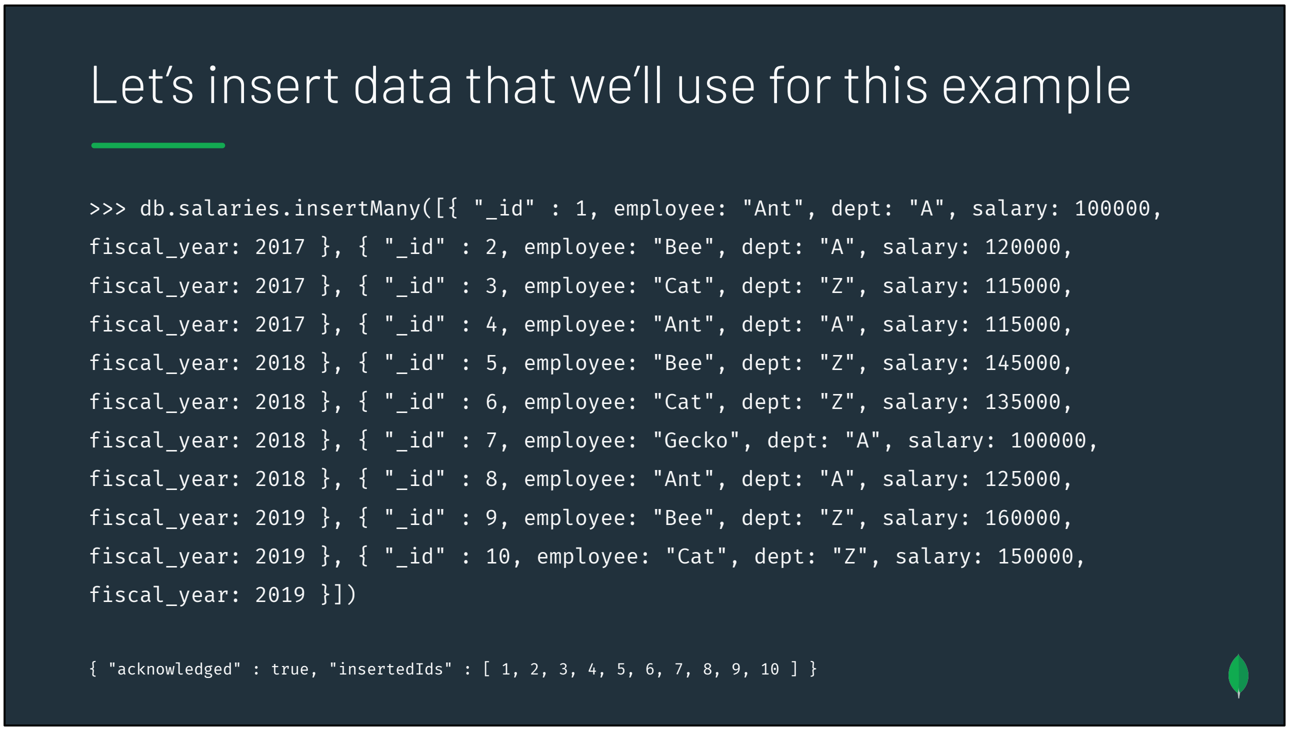
## Let’s clean up any existing data to avoid confusion

```
>>> use test
>>> db.salaries.drop()
```

Let’s first drop any previous version of the collection that might exist in our database. This is in order to avoid confusion we’ll clean any existing data so that everything starts from the same state.

We drop the collection (db.salaries.drop()) to simplify this example as existing data may change the number of documents that could be returned and it’s easier for this example to start fresh.

```
use test;
db.salaries.drop();
```



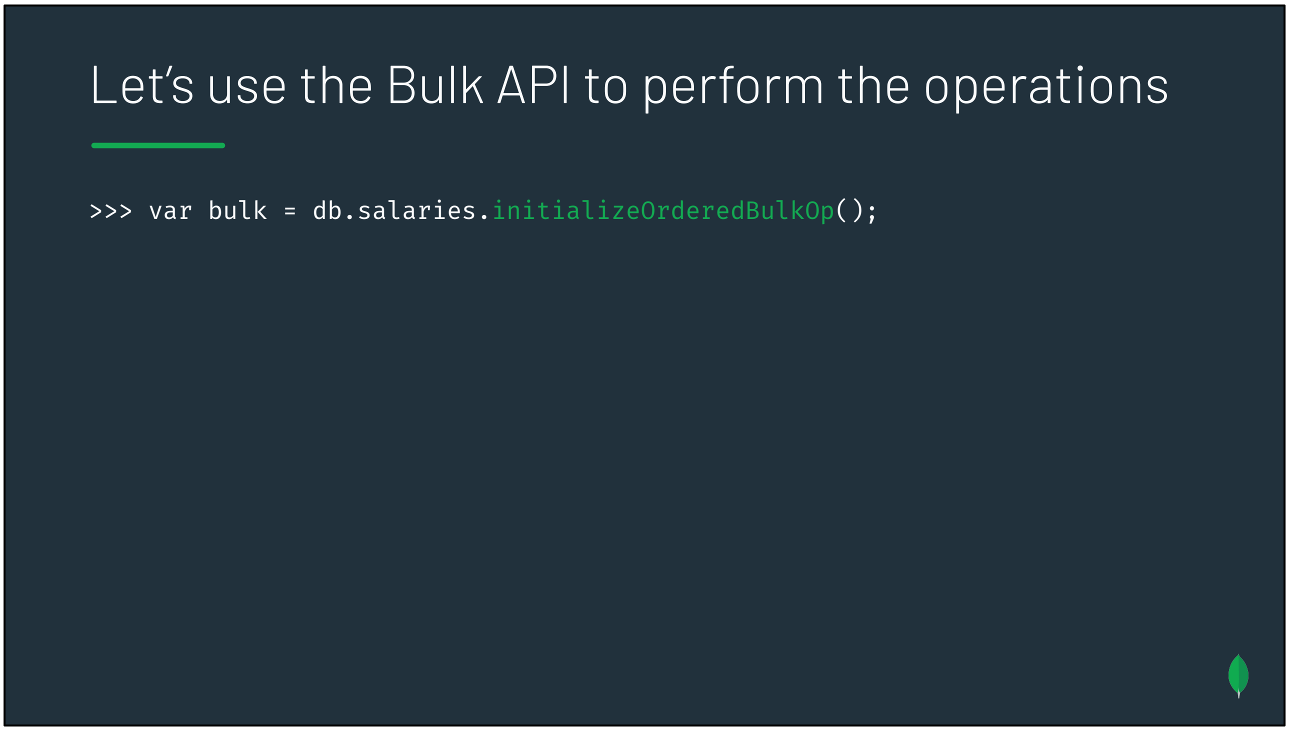
## Let’s insert data that we’ll use for this example

```
>>> db.salaries.insertMany([{ "_id" : 1, employee: "Ant", dept: "A", salary: 100000,
fiscal_year: 2017 }, { "_id" : 2, employee: "Bee", dept: "A", salary: 120000,
fiscal_year: 2017 }, { "_id" : 3, employee: "Cat", dept: "Z", salary: 115000,
fiscal_year: 2017 }, { "_id" : 4, employee: "Ant", dept: "A", salary: 115000,
fiscal_year: 2018 }, { "_id" : 5, employee: "Bee", dept: "Z", salary: 145000,
fiscal_year: 2018 }, { "_id" : 6, employee: "Cat", dept: "Z", salary: 135000,
fiscal_year: 2018 }, { "_id" : 7, employee: "Gecko", dept: "A", salary: 100000,
fiscal_year: 2018 }, { "_id" : 8, employee: "Ant", dept: "A", salary: 125000,
fiscal_year: 2019 }, { "_id" : 9, employee: "Bee", dept: "Z", salary: 160000,
fiscal_year: 2019 }, { "_id" : 10, employee: "Cat", dept: "Z", salary: 150000,
fiscal_year: 2019 }])

{ "acknowledged" : true, "insertedIds" : [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ] }
```

Let’s add ten sample documents which we’ll use for this example:

```
db.salaries.drop();
db.salaries.insertMany([{ "_id" : 1, employee: "Ant", dept:
"A", salary: 100000, fiscal_year: 2017 }, { "_id" : 2,
employee: "Bee", dept: "A", salary: 120000, fiscal_year: 2017
}, { "_id" : 3, employee: "Cat", dept: "Z", salary: 115000,
fiscal_year: 2017 }, { "_id" : 4, employee: "Ant", dept: "A",
salary: 115000, fiscal_year: 2018 }, { "_id" : 5, employee:
"Bee", dept: "Z", salary: 145000, fiscal_year: 2018 }, { "_id"
: 6, employee: "Cat", dept: "Z", salary: 135000, fiscal_year:
2018 }, { "_id" : 7, employee: "Gecko", dept: "A", salary:
100000, fiscal_year: 2018 }, { "_id" : 8, employee: "Ant",
dept: "A", salary: 125000, fiscal_year: 2019 }, { "_id" : 9,
employee: "Bee", dept: "Z", salary: 160000, fiscal_year: 2019
}, { "_id" : 10, employee: "Cat", dept: "Z", salary: 150000,
fiscal_year: 2019 }]);
```



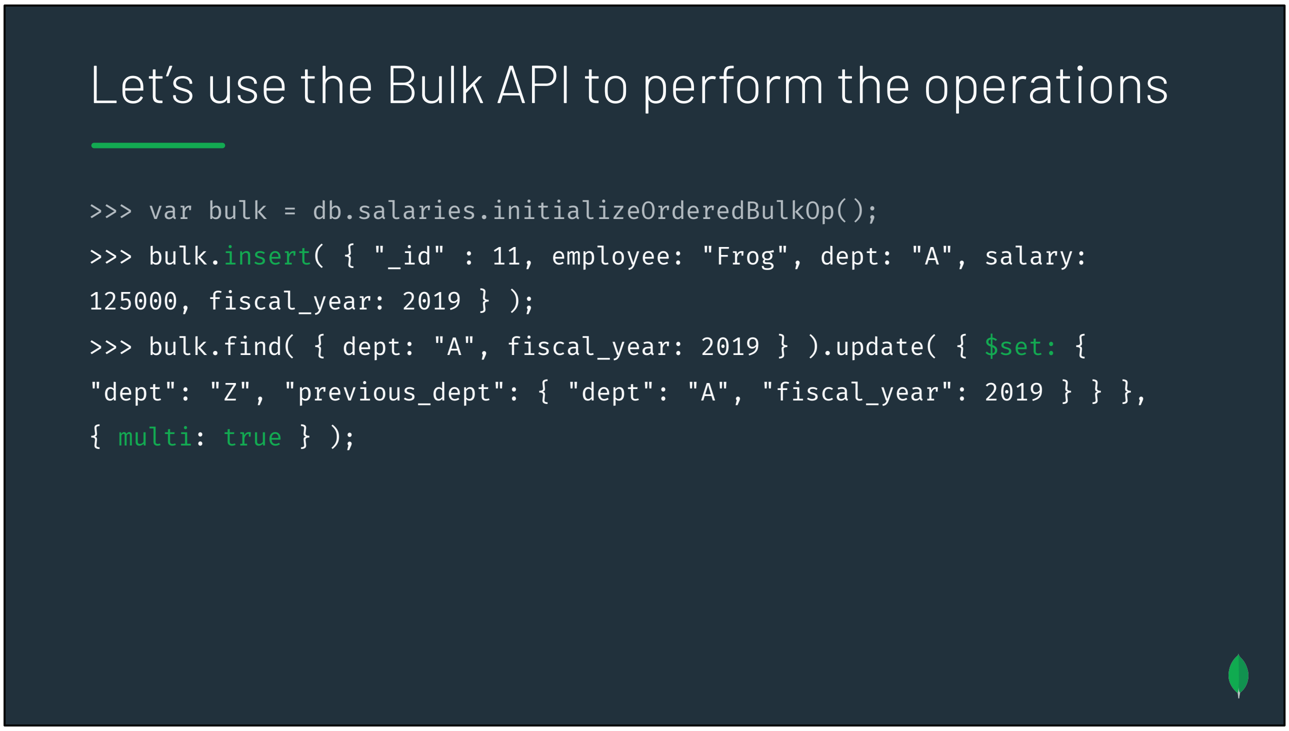
## Let’s use the Bulk API to perform the operations

```
>>> var bulk = db.salaries.initializeOrderedBulkOp();
```

Let’s just run through an example of how we can update many documents at once with the bulk api, in this example we’ll also add a document that we will later update. This is the main reason to use the ordered bulk operations so that we can be certain the document has been inserted before we begin any update operations.

Firstly, let's create the list that will hold all of the bulk operations. We'll use the initializeOrderedBulkOp as we want these operations to be processed sequentially on the database.
Here is the code block:

```
var bulk = db.salaries.initializeOrderedBulkOp();
```



## Let’s use the Bulk API to perform the operations

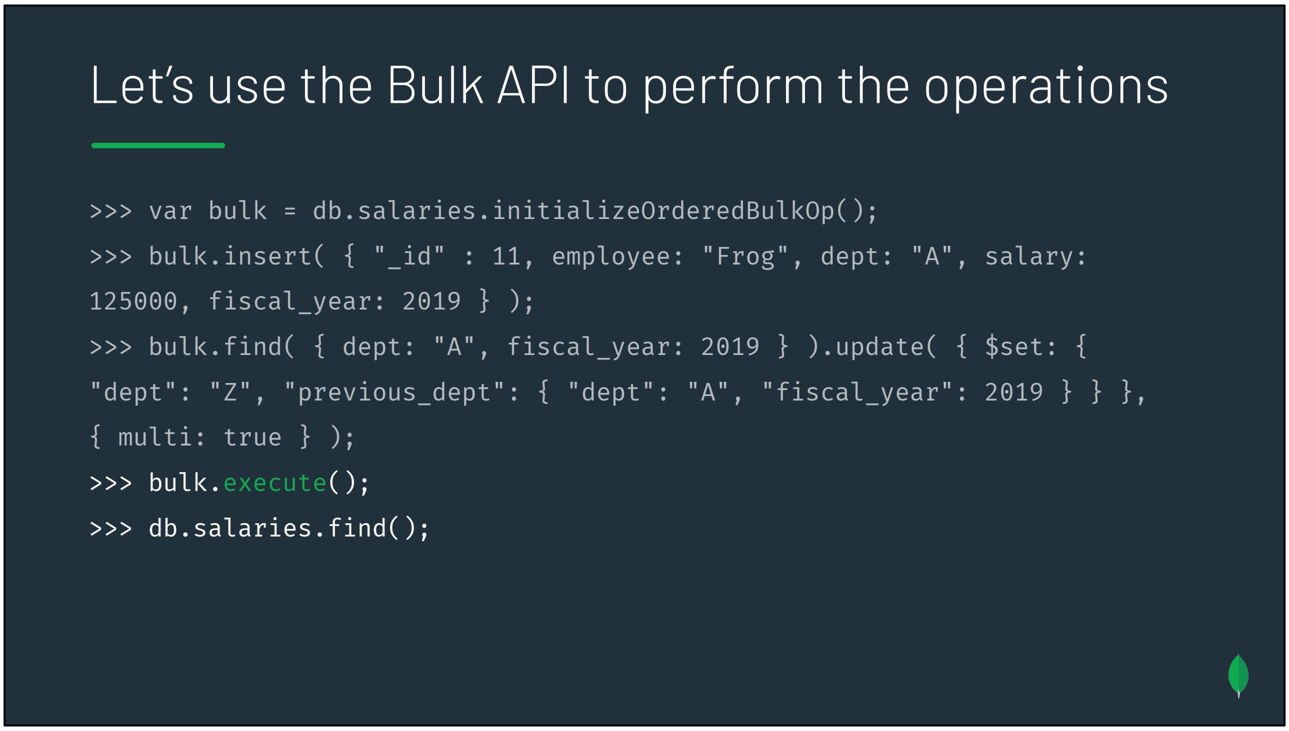
```
>>> var bulk = db.salaries.initializeOrderedBulkOp();
>>> bulk.insert( { "_id" : 11, employee: "Frog", dept: "A", salary:
125000, fiscal_year: 2019 } );
>>> bulk.find( { dept: "A", fiscal_year: 2019 } ).update( { $set: {
"dept": "Z", "previous_dept": { "dept": "A", "fiscal_year": 2019 } } },
{ multi: true } );
```

For the next step, we will add a new document with bulk.insert.

Then we will add a find and update with the multi: true option to change all of the documents from dept A to dept Z for the fiscal year 2019, for our records we are adding a previous_dept sub-document to track this change.

Here is the code block:

```
var bulk = db.salaries.initializeOrderedBulkOp();
bulk.insert( { "_id" : 11, employee: "Frog", dept: "A", salary:
125000, fiscal_year: 2019 } );
bulk.find( { dept: "A", fiscal_year: 2019 } ).update( { $set: {
"dept": "Z", "previous_dept": { "dept": "A", "fiscal_year":
2019 } } }, { multi: true } );
```



## Let’s use the Bulk API to perform the operations

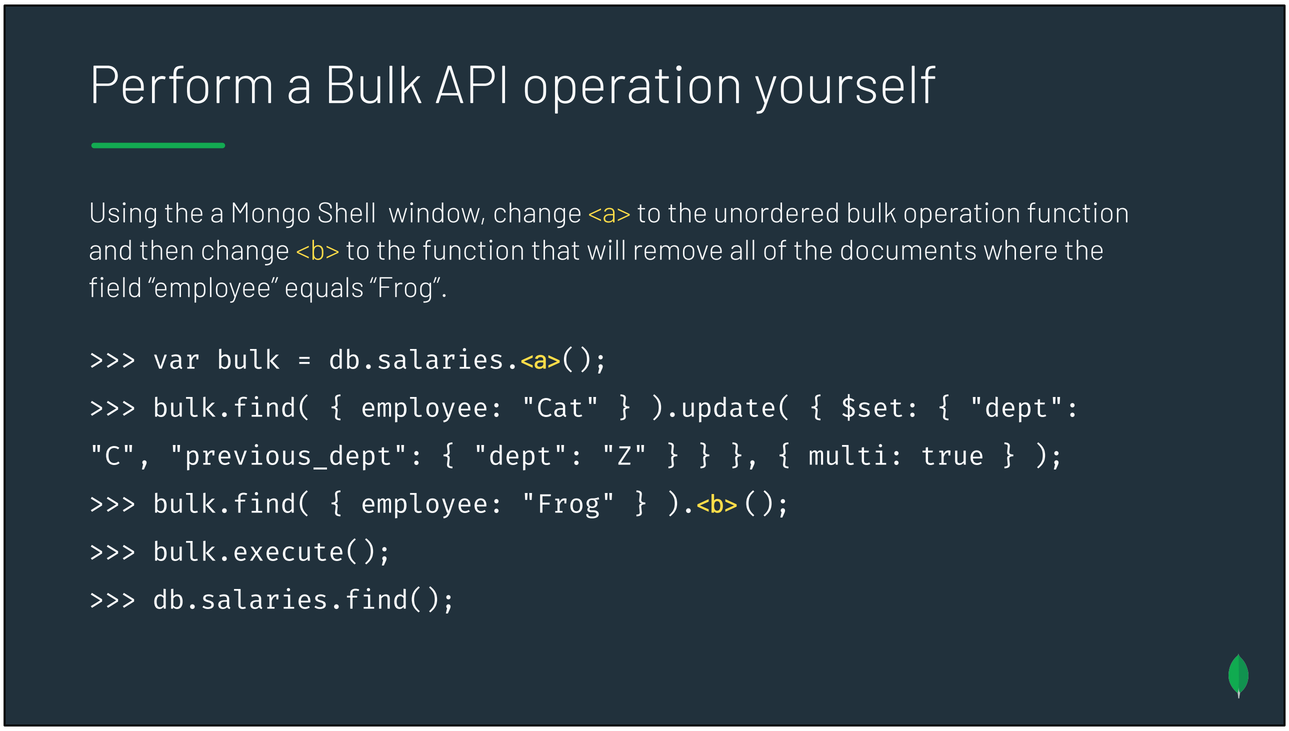
```
>>> var bulk = db.salaries.initializeOrderedBulkOp();
>>> bulk.insert( { "_id" : 11, employee: "Frog", dept: "A", salary:
125000, fiscal_year: 2019 } );
>>> bulk.find( { dept: "A", fiscal_year: 2019 } ).update( { $set: {
"dept": "Z", "previous_dept": { "dept": "A", "fiscal_year": 2019 } } },
{ multi: true } );
>>> bulk.execute();
>>> db.salaries.find();
```

Let's recap quickly, we have created the bulk variable which holds our list of two operations. The first is an insertion operation and the second is a find and update. Let's now execute these operations and see the results of the operations.

The bulk execute command will perform the operations on the database.

After this we can use a find operation on the collection to view these changes.
Here is the code block:

```
var bulk = db.salaries.initializeOrderedBulkOp();
bulk.insert( { "_id" : 11, employee: "Frog", dept: "A", salary:
125000, fiscal_year: 2019 } );
bulk.find( { dept: "A", fiscal_year: 2019 } ).update( { $set: {
"dept": "Z", "previous_dept": { "dept": "A", "fiscal_year":
2019 } } }, { multi: true } );
bulk.execute();
db.salaries.find();
```



## Perform a Bulk API operation yourself

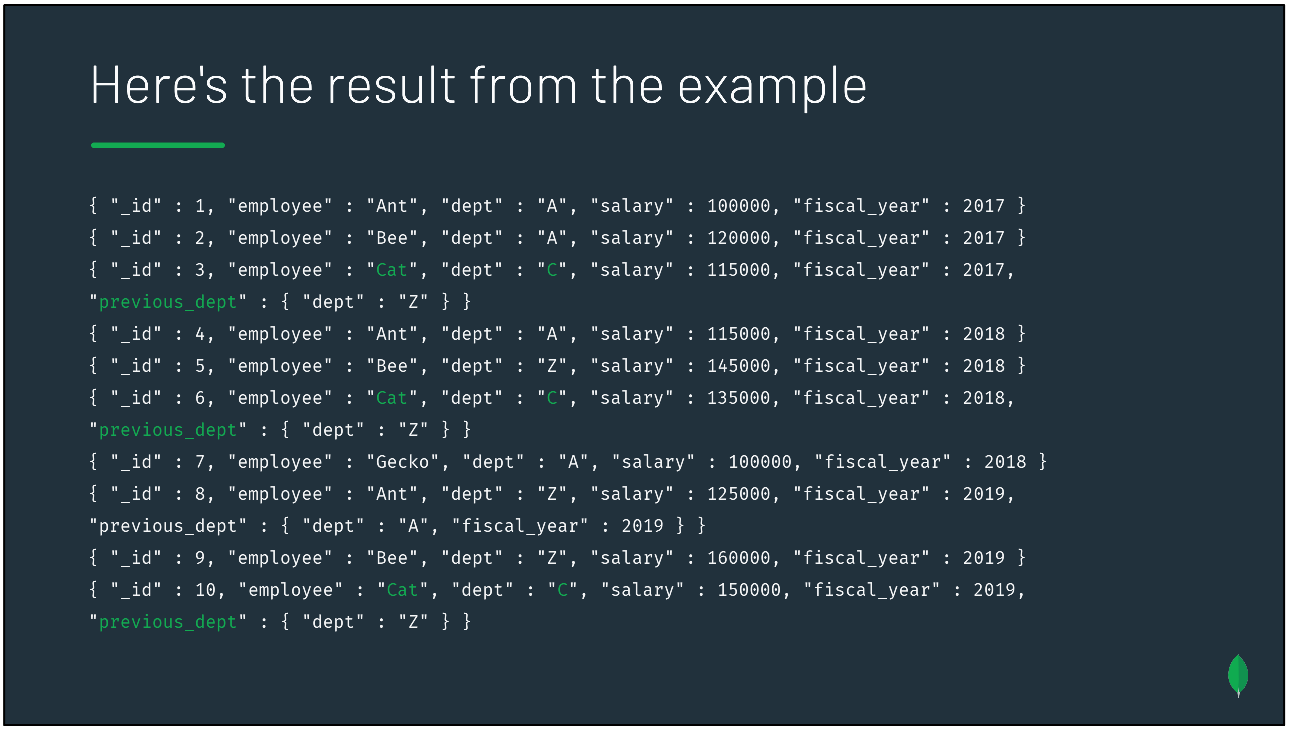
Using the a Mongo Shell  window, change <a> to the unordered bulk operation function and then change <b> to the function that will remove all of the documents where the field “employee” equals “Frog”.

```
>>> var bulk = db.salaries.<a>();
>>> bulk.find( { employee: "Cat" } ).update( { $set: { "dept":
"C", "previous_dept": { "dept": "Z" } } }, { multi: true } );
>>> bulk.find( { employee: "Frog" } ).<b>();
>>> bulk.execute();
>>> db.salaries.find();
```

Using the same MongoDB Web Shell window, change <a> to the unordered bulk operation function and then change <b> to the function that will remove all of the documents where the field “employee” equals “Frog”.

The result in the code block is what will create an unordered bulk operation that will find and update all of the documents with employee equal to “Cat” and change the department to “Z”. It should also remove all the documents with the employee equal to “Frog”.

```
var bulk = db.salaries.initializeUnorderedBulkOp();
bulk.find( { employee: "Cat" } ).update( { $set: { "dept": "C",
"previous_dept": { "dept": "Z" } } }, { multi: true } );
bulk.find( { employee: "Frog" } ).remove();
bulk.execute();
db.salaries.find();
```



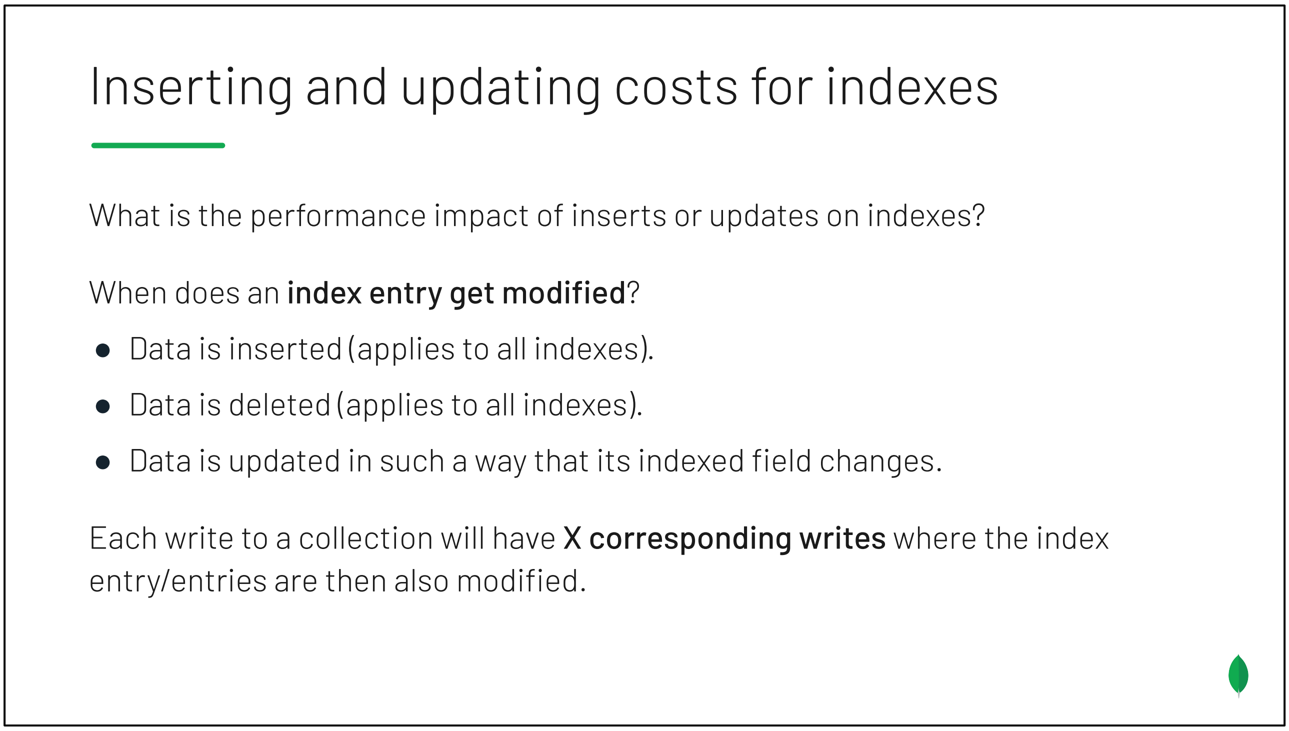
## Here's the result from the example

```
{ "_id" : 1, "employee" : "Ant", "dept" : "A", "salary" : 100000, "fiscal_year" : 2017 }
{ "_id" : 2, "employee" : "Bee", "dept" : "A", "salary" : 120000, "fiscal_year" : 2017 }
{ "_id" : 3, "employee" : "Cat", "dept" : "C", "salary" : 115000, "fiscal_year" : 2017,
"previous_dept" : { "dept" : "Z" } }
{ "_id" : 4, "employee" : "Ant", "dept" : "A", "salary" : 115000, "fiscal_year" : 2018 }
{ "_id" : 5, "employee" : "Bee", "dept" : "Z", "salary" : 145000, "fiscal_year" : 2018 }
{ "_id" : 6, "employee" : "Cat", "dept" : "C", "salary" : 135000, "fiscal_year" : 2018,
"previous_dept" : { "dept" : "Z" } }
{ "_id" : 7, "employee" : "Gecko", "dept" : "A", "salary" : 100000, "fiscal_year" : 2018 }
{ "_id" : 8, "employee" : "Ant", "dept" : "Z", "salary" : 125000, "fiscal_year" : 2019,
"previous_dept" : { "dept" : "A", "fiscal_year" : 2019 } }
{ "_id" : 9, "employee" : "Bee", "dept" : "Z", "salary" : 160000, "fiscal_year" : 2019 }
{ "_id" : 10, "employee" : "Cat", "dept" : "C", "salary" : 150000, "fiscal_year" : 2019,
"previous_dept" : { "dept" : "Z" } }
```

Here are the results from the Bulk API example you have just run, we can see the updates made ot all the "Cat" employee documents and that there are no documents with "Frog" in the employee field remaining.

## Impact on indexes

Let’s recap a few points from our earlier lesson on indexing in terms of the impact of insertions and updates of data and what this entails for indexes and performance.



## Inserting and updating costs for indexes

What is the performance impact of inserts or updates on indexes?

When does an **index entry get modified**?

- Data is inserted (applies to all indexes).
- Data is deleted (applies to all indexes).
- Data is updated in such a way that its indexed field changes.

Each write to a collection will have **X corresponding writes** where the index entry/entries are then also modified.

What is the performance impact of inserts or updates on indexes? An index gets modified every time a field that is an index key is modified. This means it will also need a write operation.

When does an index entry get modified?

An index is modified any time a document:
- Is inserted (applies to all indexes)
- Is deleted (applies to all indexes)
- Is updated in such a way that its indexed field changes

Each write operation (insert/delete/update to indexed field) will result in X corresponding writes needing to be made to ensure each of the applicable index entries are also updated to reflect the change.

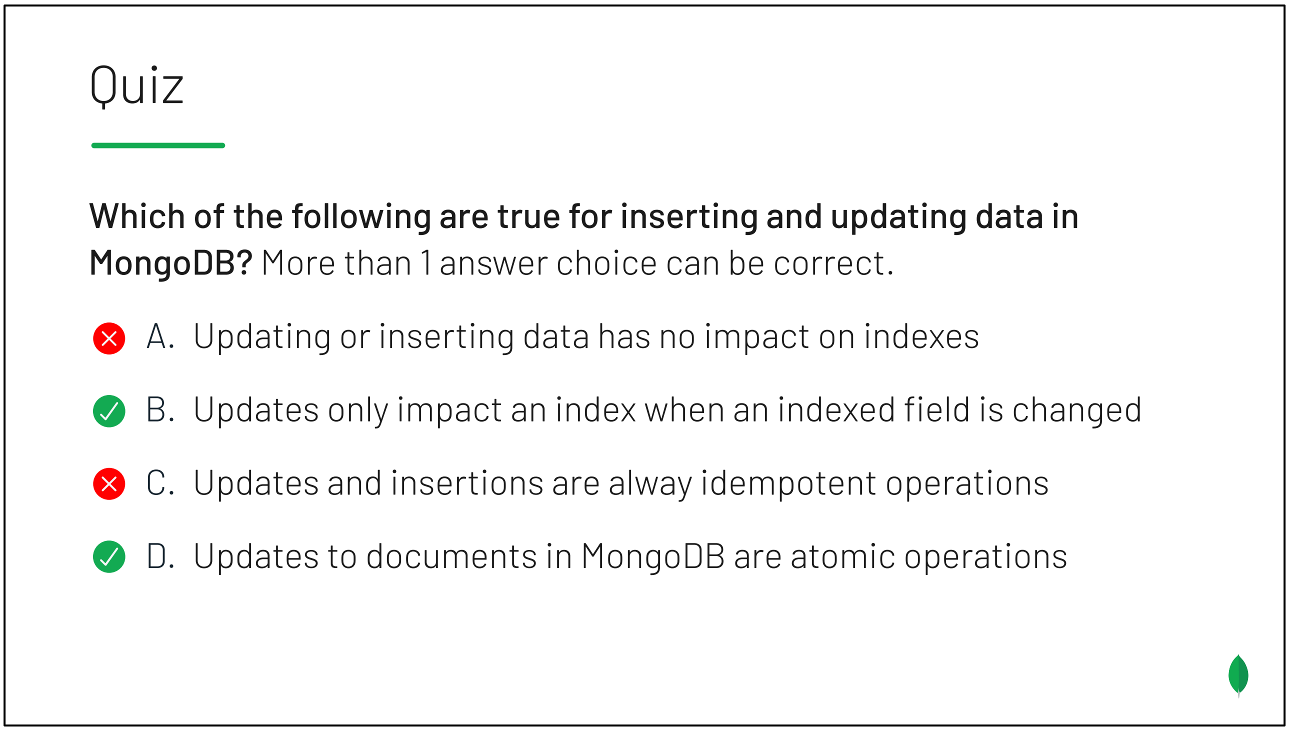
In the case of a collection with four indexes where a new document is inserted then five write operations will occur, the new document will be written to the collection and each of the four indexes will also be modified to include index entries for the new document.

## Quiz

## Quiz

**Which of the following are true for inserting and updating data in MongoDB?** More than 1 answer choice can be correct.

- A. Updating or inserting data has no impact on indexes
- B. Updates only impact an index when an indexed field is changed
- C. Updates and insertions are alway idempotent operations
- D. Updates to documents in MongoDB are atomic operations

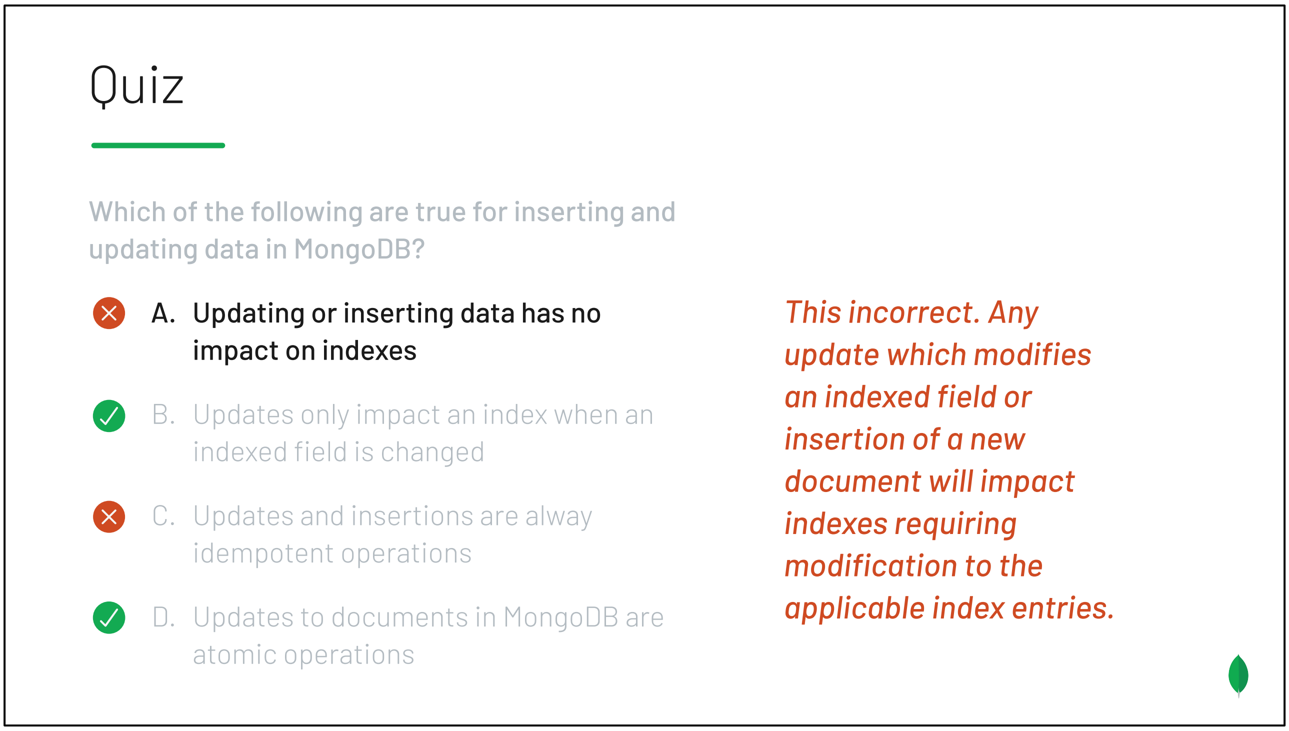


## Quiz

**Which of the following are true for inserting and updating data in MongoDB?** More than 1 answer choice can be correct.

- ✗ A. Updating or inserting data has no impact on indexes
- ✓ B. Updates only impact an index when an indexed field is changed
- ✗ C. Updates and insertions are alway idempotent operations
- ✓ D. Updates to documents in MongoDB are atomic operations

INCORRECT: Updating or inserting data has no impact on indexes - Any update which modifies an indexed field or insertion of a new document will impact indexes requiring modification to the applicable index entries.
CORRECT: Updates only impact an index when an indexed field is changed - Changes to unindexed fields do not cause an impact to the existing index entries.
INCORRECT: Updates and insertions are alway idempotent operations - Insertion are idempotent operations, however not all updates are idempotent operations. We covered the example of the $inc operator which is not idempotent versus the $set operator, there are several other operators which are not idempotent, please refer to the MongoDB update operator documentation page for specifics.
CORRECT: Updates to documents in MongoDB are atomic operations - a write operation on a single document is atomic, so updates at the level of a document are atomic.



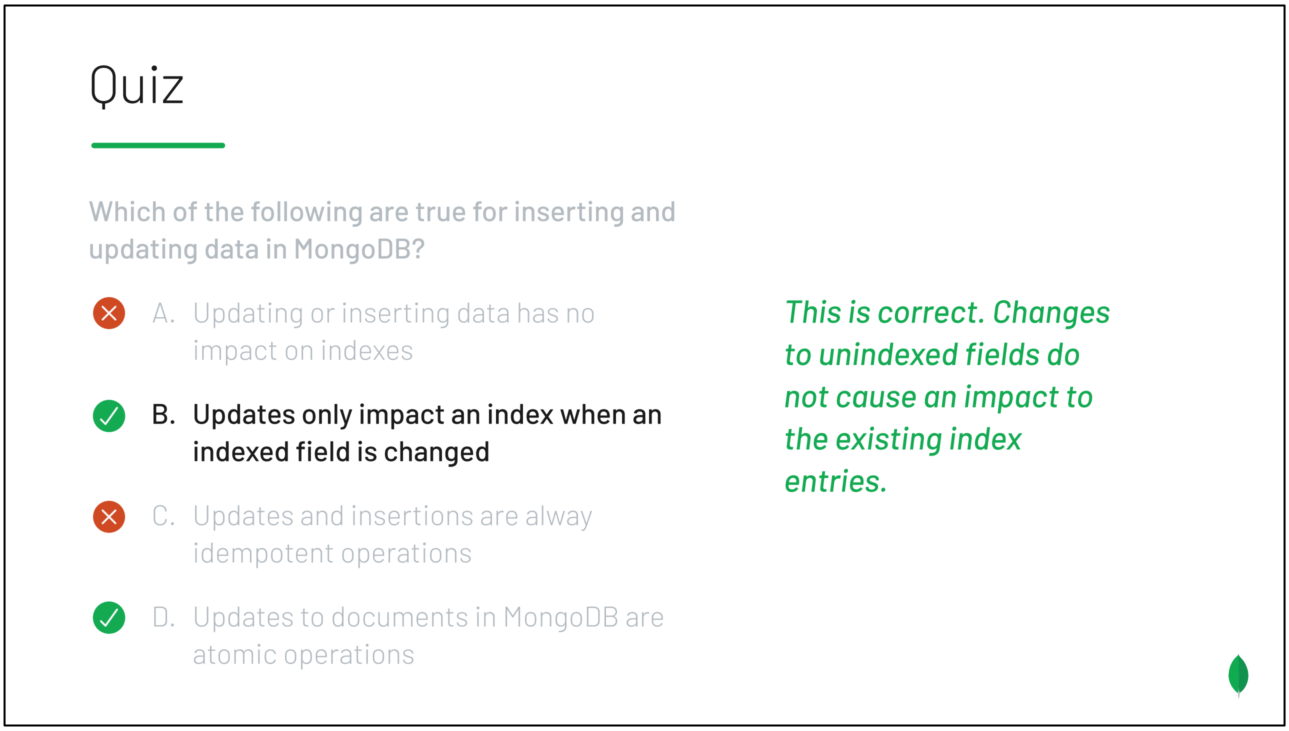
## Quiz

Which of the following are true for inserting and updating data in MongoDB?

- ✗ A. Updating or inserting data has no impact on indexes
- ✓ B. Updates only impact an index when an indexed field is changed
- ✗ C. Updates and insertions are alway idempotent operations
- ✓ D. Updates to documents in MongoDB are atomic operations

*This incorrect. Any update which modifies an indexed field or insertion of a new document will impact indexes requiring modification to the applicable index entries.*

INCORRECT: Updating or inserting data has no impact on indexes - This is incorrect. Any update which modifies an indexed field or insertion of a new document will impact indexes requiring modification to the applicable index entries.



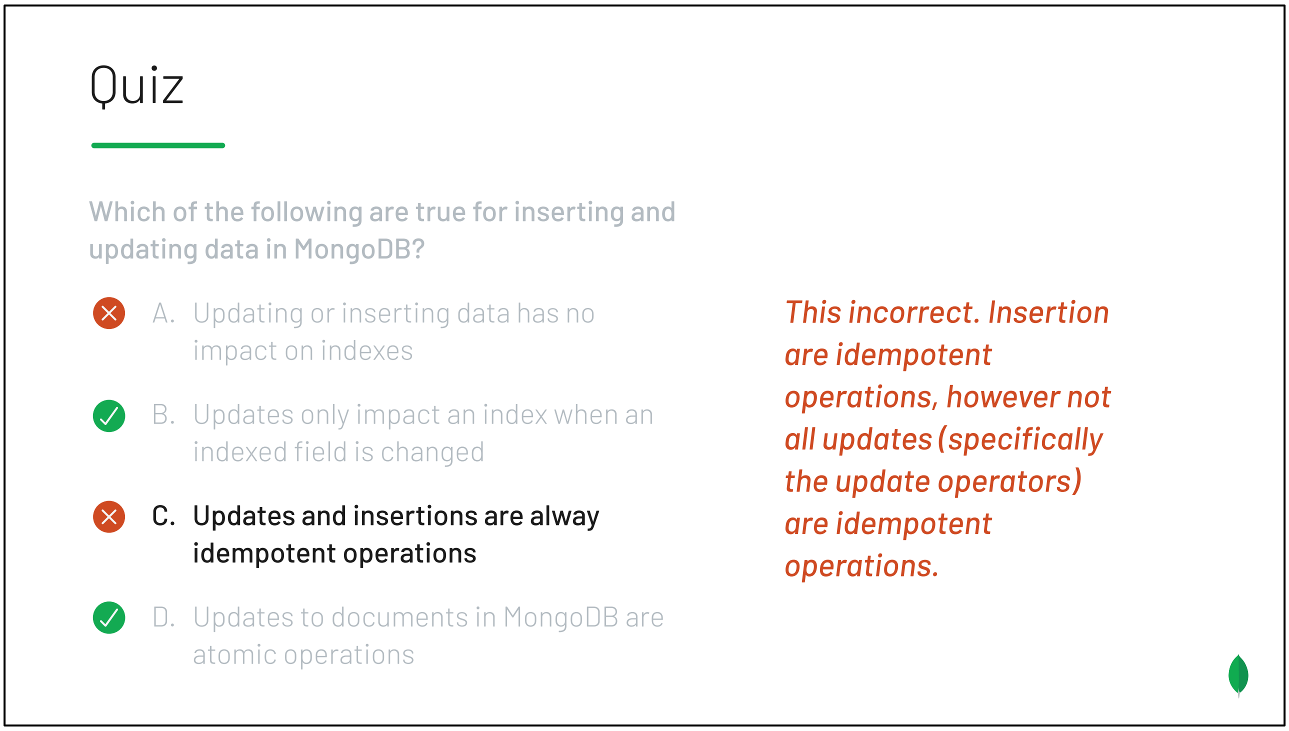
## Quiz

Which of the following are true for inserting and updating data in MongoDB?

- ✗ A. Updating or inserting data has no impact on indexes
- ✓ B. Updates only impact an index when an indexed field is changed
- ✗ C. Updates and insertions are alway idempotent operations
- ✓ D. Updates to documents in MongoDB are atomic operations

*This is correct. Changes to unindexed fields do not cause an impact to the existing index entries.*

CORRECT: Updates only impact an index when an indexed field is changed - This is correct. Changes to unindexed fields do not cause an impact to the existing index entries.



## Quiz

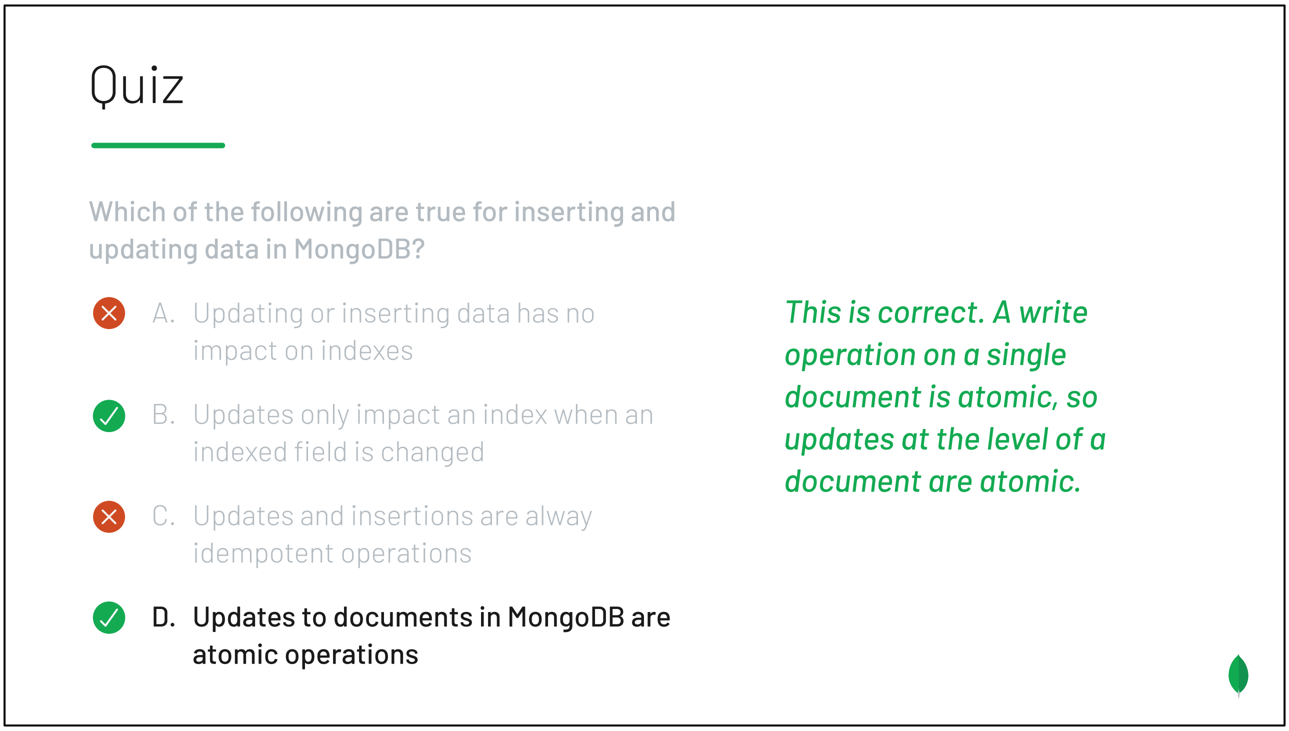
Which of the following are true for inserting and updating data in MongoDB?

- ✗ A. Updating or inserting data has no impact on indexes
- ✓ B. Updates only impact an index when an indexed field is changed
- ✗ C. **Updates and insertions are alway idempotent operations**
- ✓ D. Updates to documents in MongoDB are atomic operations

*This incorrect. Insertion are idempotent operations, however not all updates (specifically the update operators) are idempotent operations.*

INCORRECT: Updates and insertions are alway idempotent operations - This is incorrect. Insertion are idempotent operations, however not all updates (specifically the update operators) are idempotent operations.

Note: We covered the example of the $inc operator which is not idempotent versus the $set operator, there are several other operators which are not idempotent, please refer to the MongoDB update operator documentation page for specifics.



## Quiz

Which of the following are true for inserting and updating data in MongoDB?

✗ A. Updating or inserting data has no impact on indexes

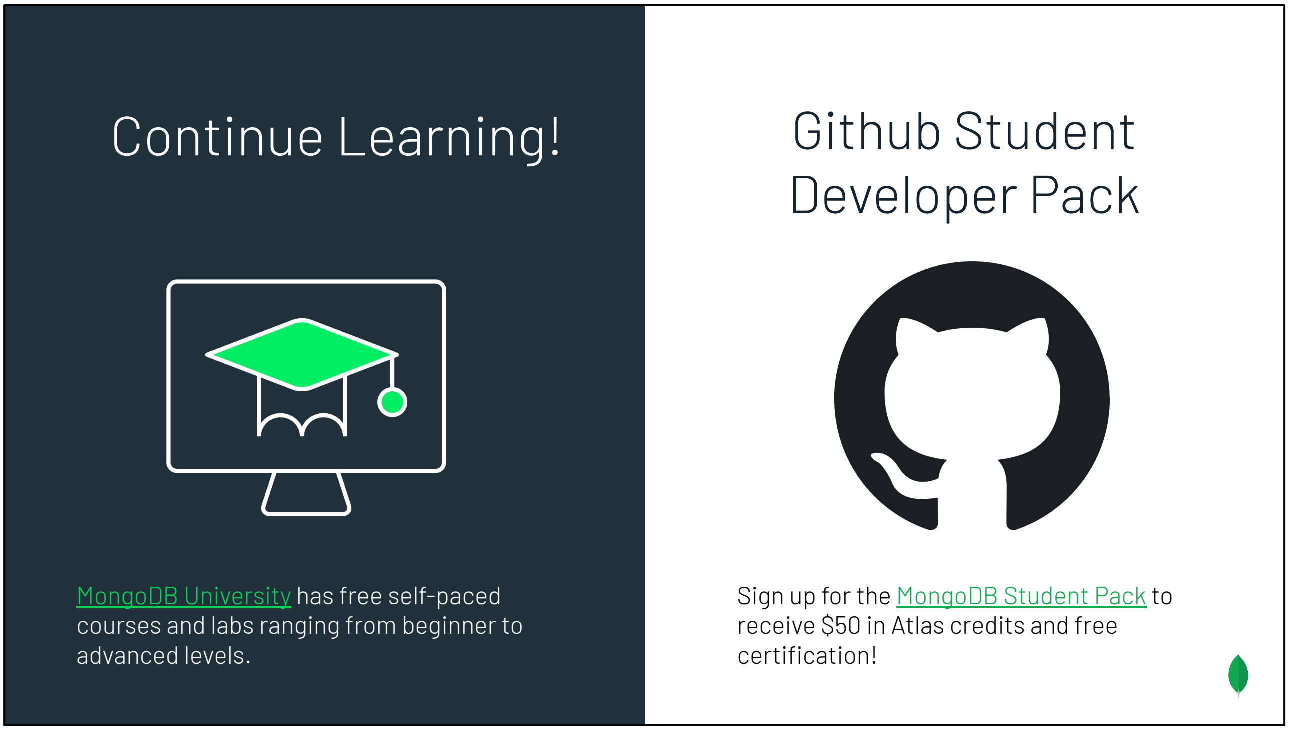
✓ B. Updates only impact an index when an indexed field is changed

✗ C. Updates and insertions are alway idempotent operations

✓ **D. Updates to documents in MongoDB are atomic operations**

*This is correct. A write operation on a single document is atomic, so updates at the level of a document are atomic.*

CORRECT: Updates to documents in MongoDB are atomic operations - This is correct. A write operation on a single document is atomic, so updates at the level of a document are atomic.



## Continue Learning!

MongoDB University has free self-paced courses and labs ranging from beginner to advanced levels.

## Github Student Developer Pack

Sign up for the MongoDB Student Pack to receive $50 in Atlas credits and free certification!

This concludes the material for this lesson. However, there are many more ways to learn about MongoDB and non-relational databases, and they are all free! Check out MongoDB's University page to find free courses that go into more depth about everything MongoDB and non-relational. For students and educators alike, MongoDB for Academia is here to offer support in many forms. Check out our educator resources and join the Educator Community. Students can receive $50 in Atlas credits and free certification through the Github Student Developer Pack.