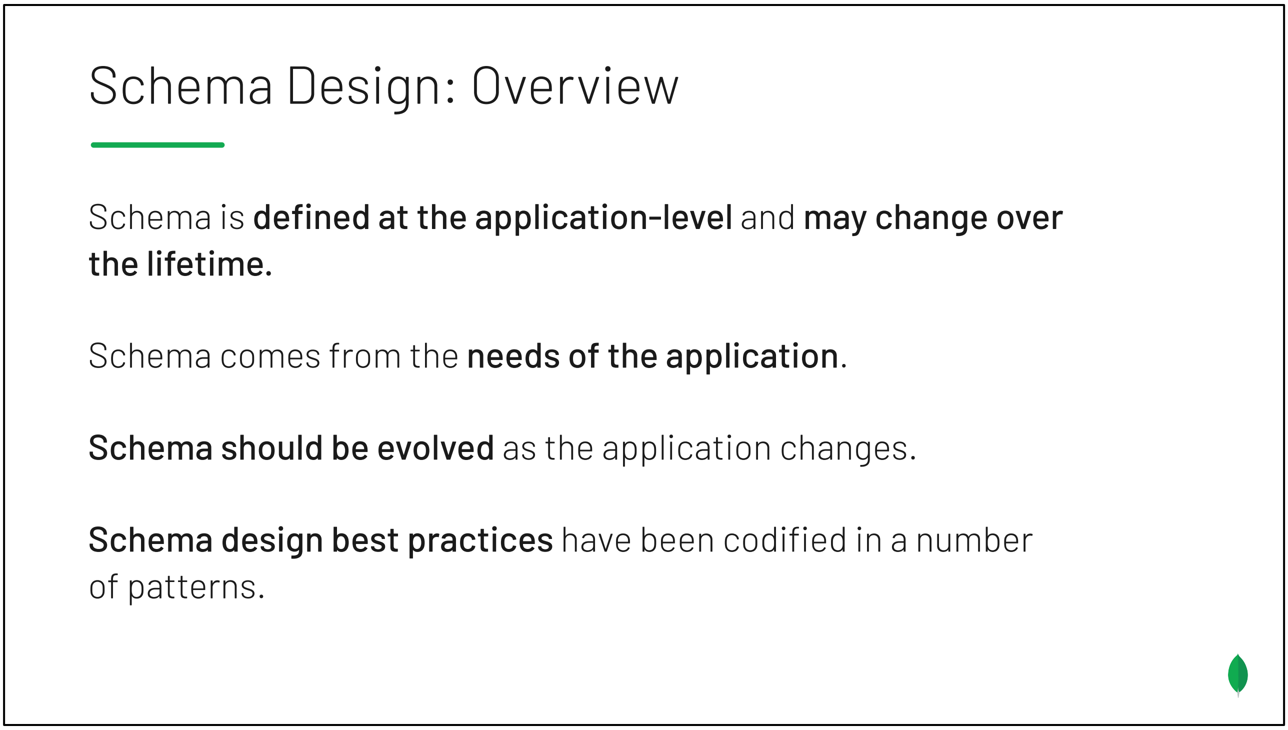
mongoDB®

Data Modeling and Schema Design Patterns

Google slide deck available here



Schema Design: Overview

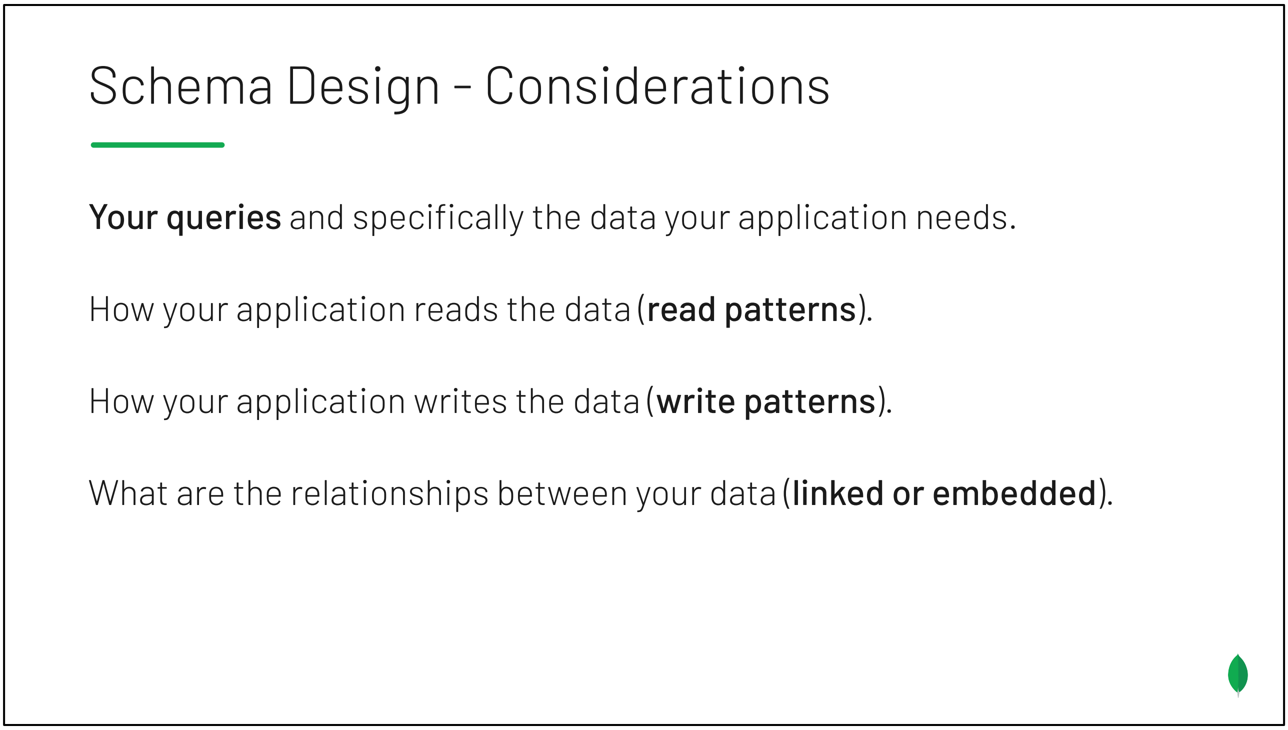
Schema is **defined at the application-level** and **may change over the lifetime.**

Schema comes from the **needs of the application**.

Schema should be evolved as the application changes.

Schema design best practices have been codified in a number of patterns.

Schema design is defined at the application level and it is likely it will change over the application’s lifetime.
The schema should be designed to service the needs of the application.
It will likely evolve as the application changes.
Schema design patterns are a codified set of approached to schema design which help you use best practices within your schema.



Schema Design - Considerations

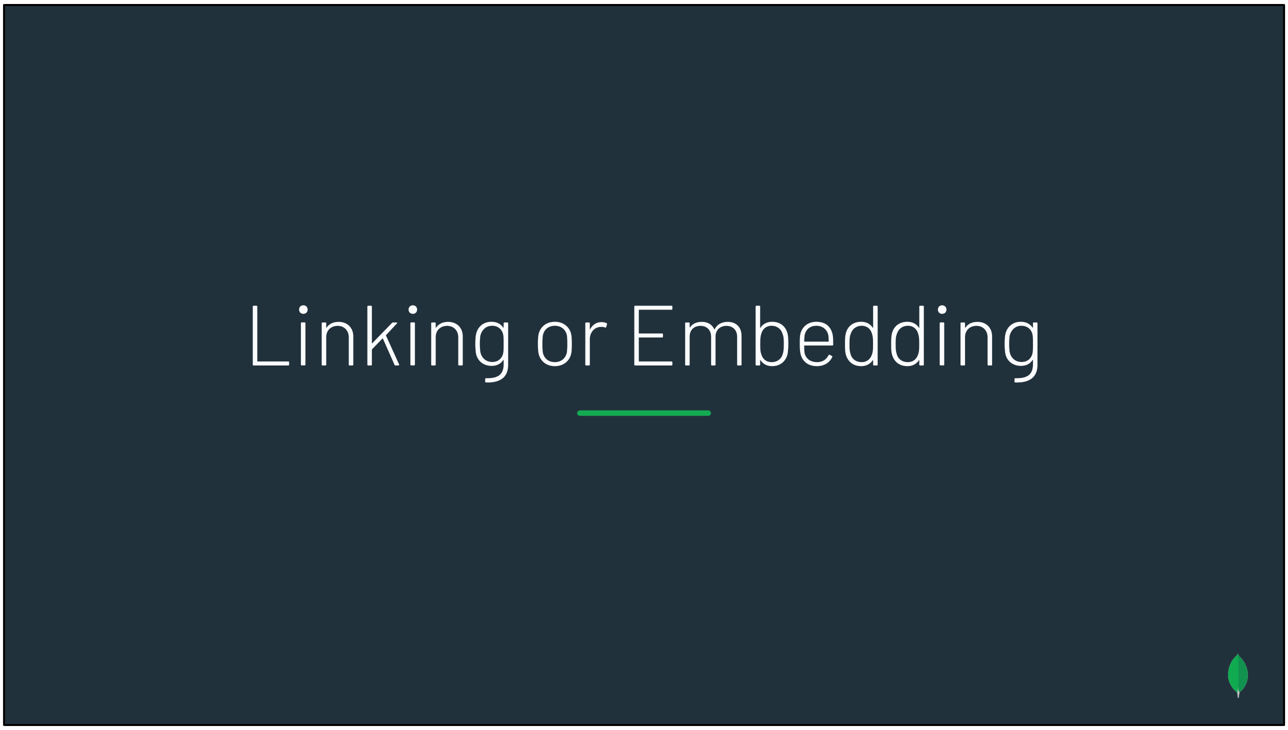
Your queries and specifically the data your application needs.

How your application reads the data (**read patterns**).

How your application writes the data (**write patterns**).

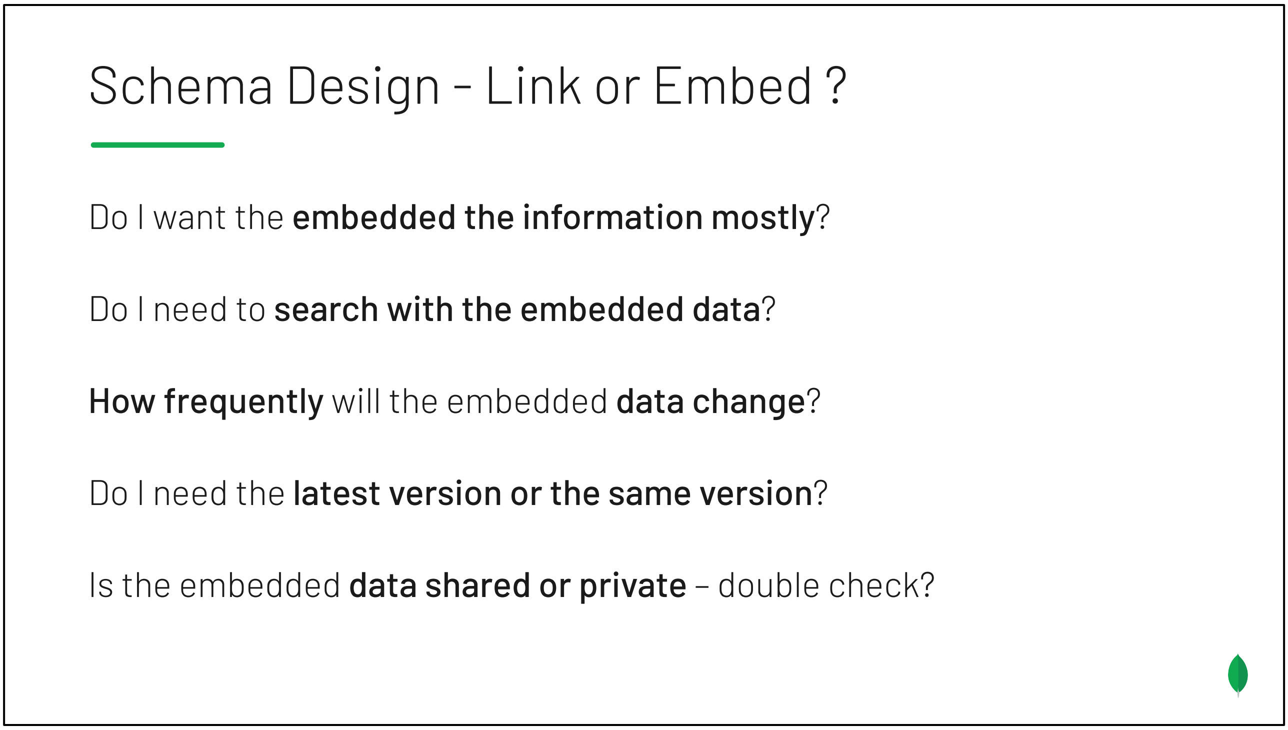
What are the relationships between your data (**linked or embedded**).

There are a number of considerations you should focus on within schema design.
Specifically, you need to understand the data and the queries needed by your application.
You also need to understand how it reads and writes data.
Finally, you also need to understand the relationships between your data.
We’ll cover a methodology later in this lesson that should help you ask the right questions to create a schema that fits these considerations.



Linking or Embedding

Let’s look at linking data in documents across multiple collections and let’s also look at embedding data in one document these are the main approaches to modelling data in MongoDB.



Schema Design - Link or Embed ?

Do I want the **embedded the information mostly**?

Do I need to **search with the embedded data**?

How frequently will the embedded **data change**?

Do I need the **latest version or the same version**?

Is the embedded **data shared or private** – double check?

When you are performing schema design the question of whether to link or to embed your data are at the core of any schema design.
The questions here should all be answered when you are creating your schema design as they can help ensure you fully consider all aspects of the design.

Firstly, Do I want the embed the information mostly ? In this case, if all the information is present in a single document then retrieval will be faster than if it is in two documents or across two or more collections. The frequency of the specific data and the size of the data must also be considered as very large documents will not be retrieved as fast as tiny documents even if those documents are in two separate collections.

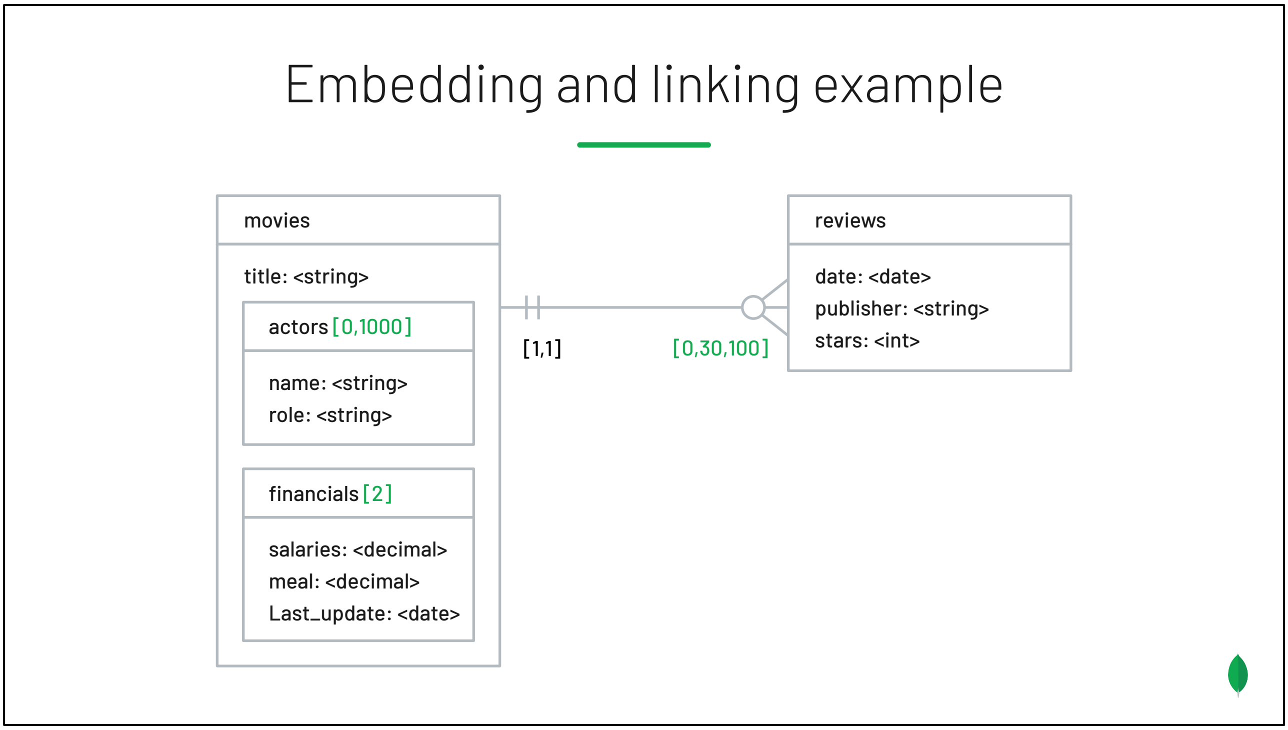
A second important question to ask is Do I need to search with the embedded data? You might also want to follow up to determine what you are searching for and if it is only a fraction of the embedded data, should only some of it be embedded and the rest linked.

Thirdly, you should ask How frequently will the embedded data change.

Fourthly, you should consider whether the latest schema version or the same version of the schema is required. The answer of this would indicate for instance if the schema versioning pattern discussed later in the lesson might be applicable.

Finally, you should determine whether the embedded data can be shared or whether it is private. Indeed, this should be double checked with all stakeholders to ensure there are no doubts on this question which is increasingly important in the age of both data and digital privacy.

An important set of questions that should be answered when designing your schema is whether you should link to other documents with the data or whether the data should be embedded in a single document. This should also be considered in terms of the different users of the database as this might not be the same answer depending on the specific user.



Embedding and linking example

movies

title: <string>

actors [0,1000]

name: <string>
role: <string>

financials [2]

salaries: <decimal>
meal: <decimal>
Last_update: <date>

[1,1]

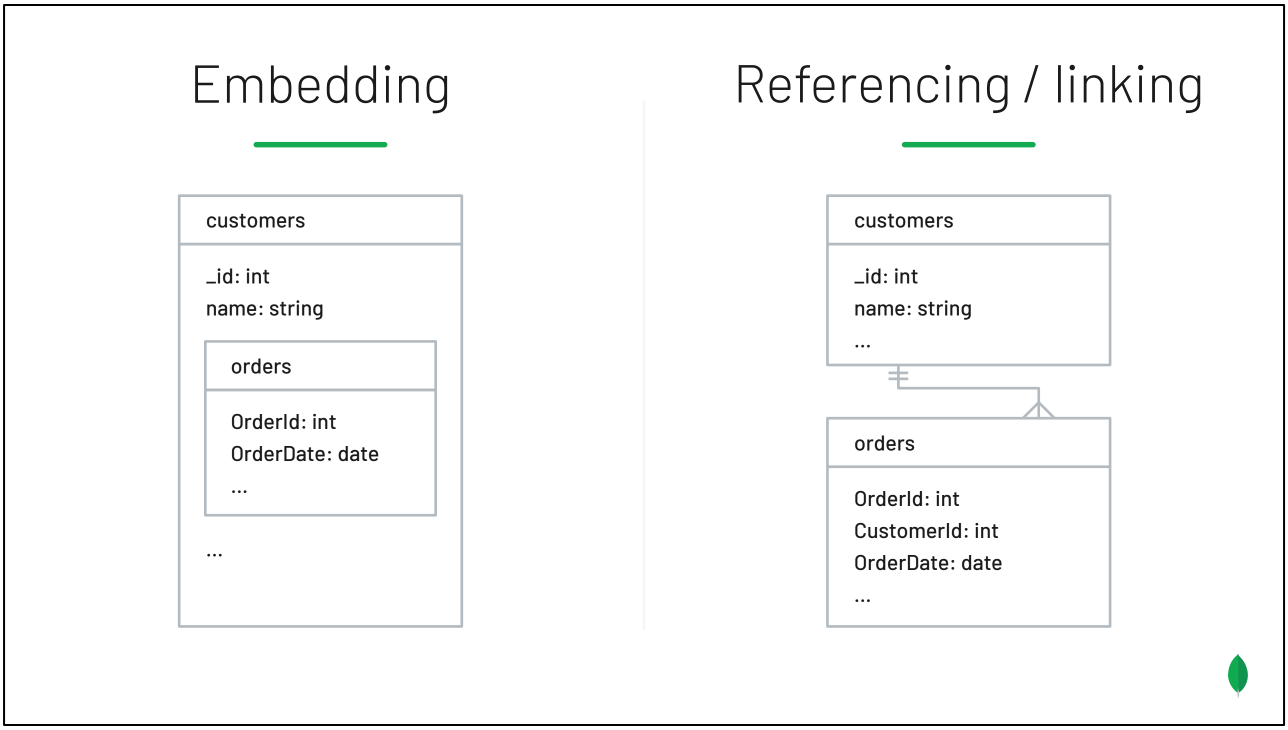
[0,30,100]

reviews

date: <date>
publisher: <string>
stars: <int>

Let’s look at an application where we store details on movies and reviews related to those movies.

In this example, we have a movie document where the actors and the financials are embedded in a single document whilst reviews are linked.



Embedding

customers

_id: int
name: string

orders

OrderId: int
OrderDate: date
...

...

Referencing / linking

customers

_id: int
name: string
...

orders

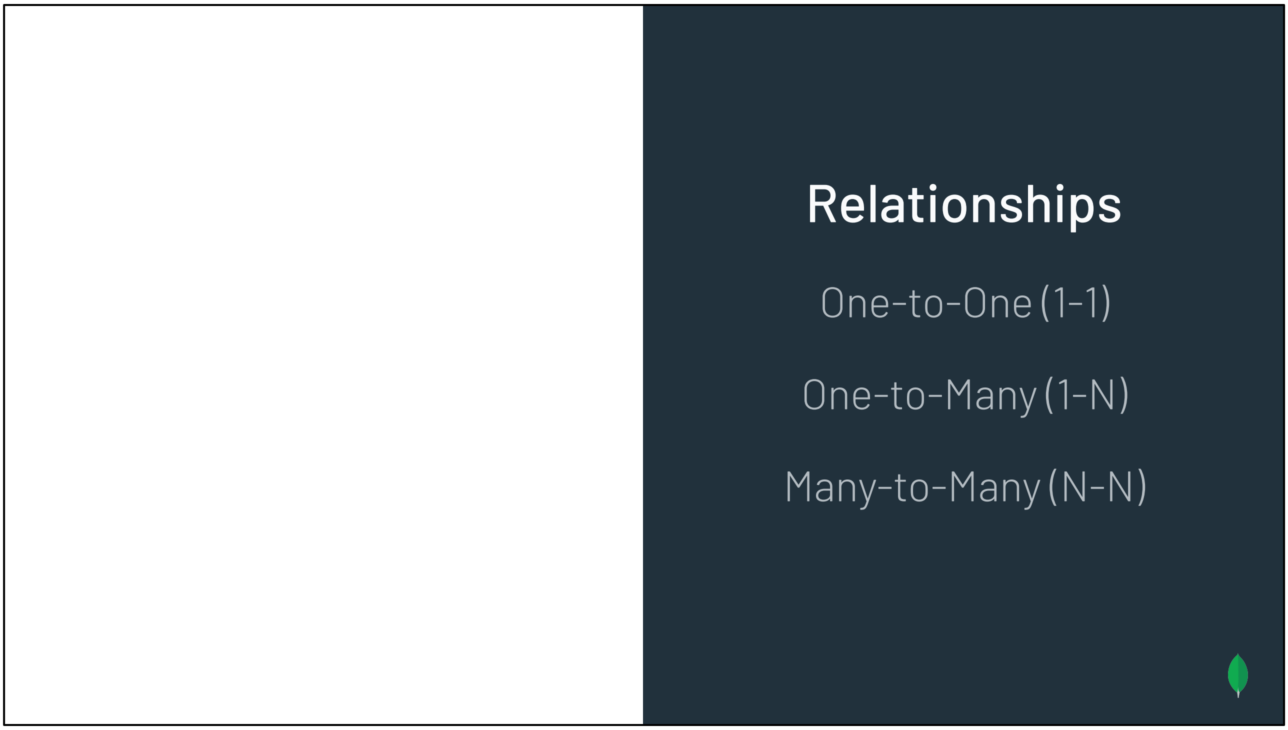
OrderId: int
CustomerId: int
OrderDate: date
...

Let’s look at another scenario where we want to store customers and their related order data.
We can do this with either embedding (on the left) or linking (on the right) as shown on the slide.

It is equally possible to design with either approach that is to say in this example on the slide you could either link to the orders for a customer which are kept in a separate orders table or they could equally be embedded within the customers document.

In general, the rule of thumb is to favour embedding over linking. This typically gives the required data with a single query. It is also a better design when data that will be deleted exists together.

This all depends on the situation and if there is a large amount of unused data in the document or say only the last 20 orders are required then it might be considering an alternative design. We’ll cover one schema design pattern later in this lesson, the subset pattern which would be appropriate for that situation.



Relationships

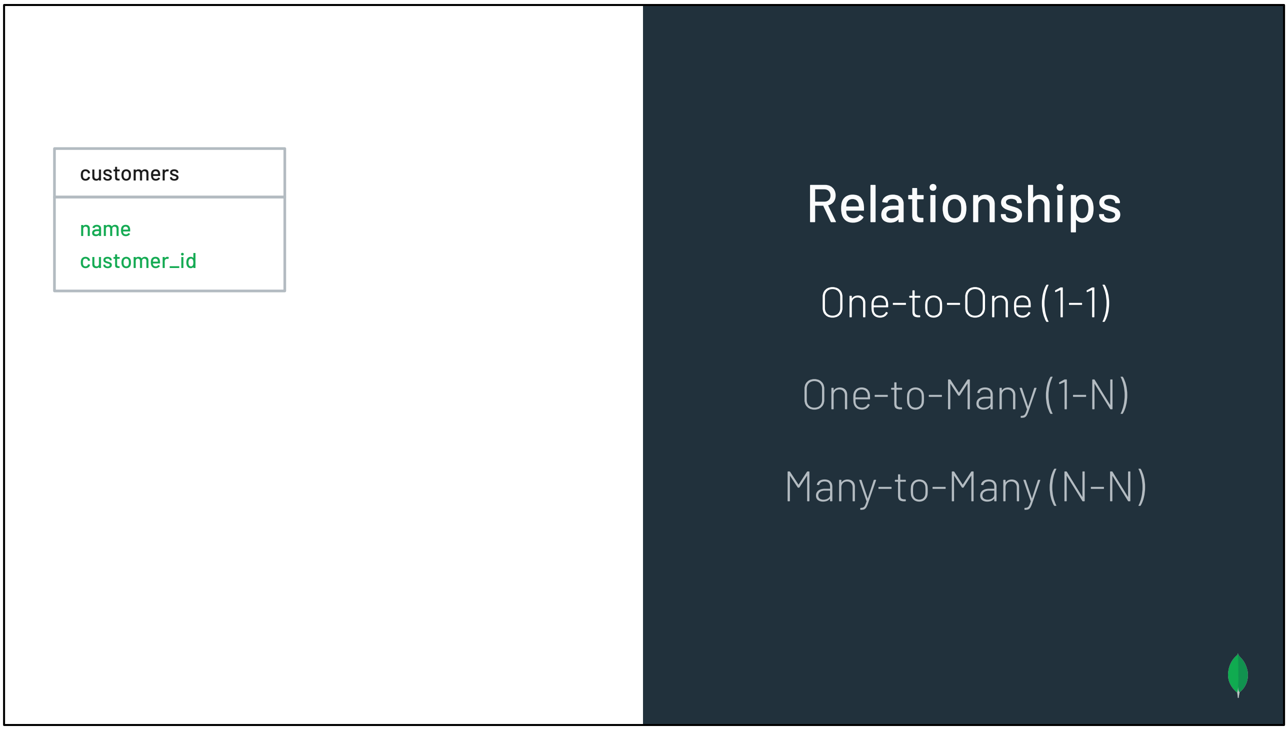
One-to-One (1-1)

One-to-Many (1-N)

Many-to-Many (N-N)

We’re going to talk about the relationship or link types that you will typically use to model data.

We’ll look at one to one relations firstly, then we’ll move to one to many, and finally we’ll move to many to many relationships.



customers

name
customer_id

Relationships

One-to-One (1-1)

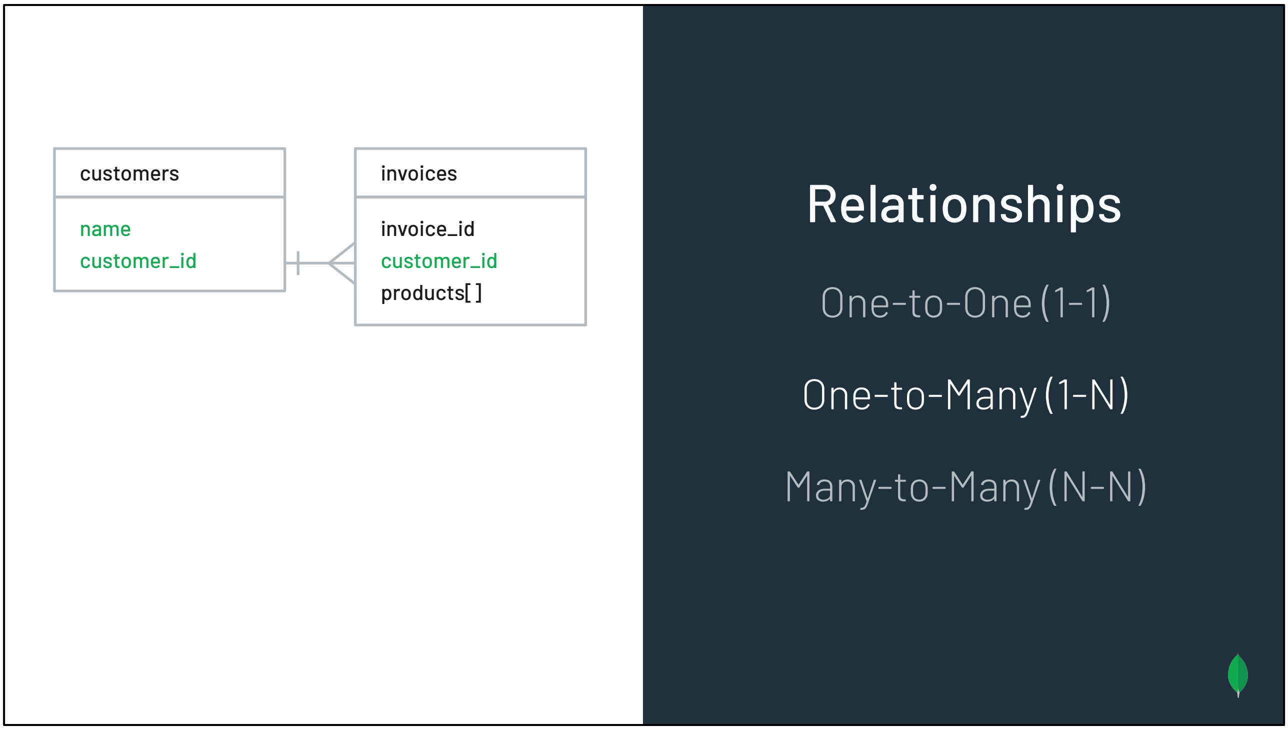
One-to-Many (1-N)

Many-to-Many (N-N)

A one-to-one relationship is represented and stored in a single document, this would typically be data like a person's name and the customer id. All these fields have a one-to-one relationship with each other. More clearly, a user in our system has one and only one name, and is associated with one and only one customer id.

In designing this relationship, if you only consider one side of the relationship you may classify it as a “one” or as a “many” if you don’t consider both sides. The best advice is to ensure you ask the question of associativity from both directions and that you review your model a few times, especially for less apparent relationships.

Embedding is the preferred way to model a 1:1 relationship as it’s more efficient to retrieve the document.



customers

name
customer_id

invoices

invoice_id
customer_id
products[ ]

Relationships

One-to-One (1-1)

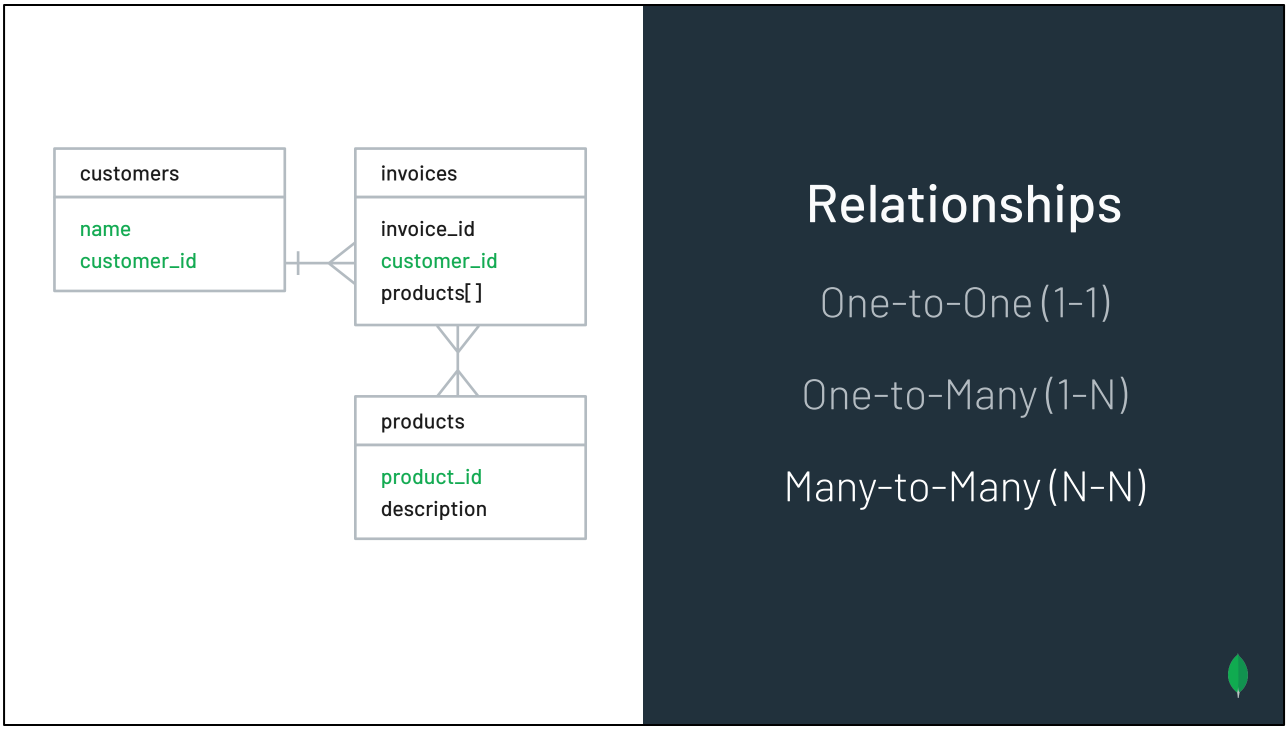
One-to-Many (1-N)

Many-to-Many (N-N)

A one-to-many relationship can be considered as that when an object of a given type is associated with N objects of a second type.

In this type of relationship, the customer can have many invoices. It can be modelled by either linking the data (as shown) or by embedding the data (where the invoices are within the customer documents).

There is an additional technical called bucketing which is a combination of both linking and embedding. Bucketing works best when you can split your documents into batches, it can speed up document retrieval. Time based data (IOT) or where there are a number of entries (e.g. like comments pagination). We'll look at bucketing in more depth as part of the schema design patterns later in this lesson.



customers

name
customer_id

invoices

invoice_id
customer_id
products[ ]

products

product_id
description

Relationships

One-to-One (1-1)

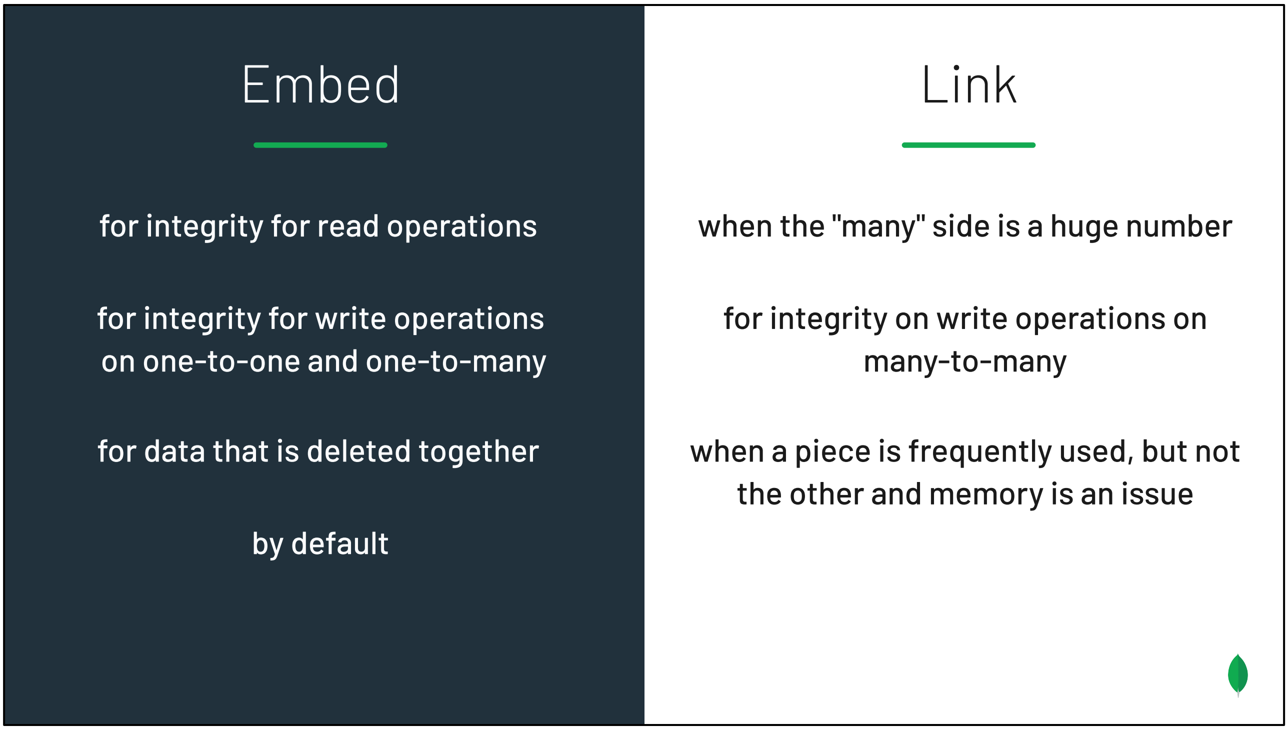
One-to-Many (1-N)

Many-to-Many (N-N)

A Many-to-Many relationship between two entities where they both might have many relationships between each other. This means that documents on the first side can be associated with many documents on the second side. In terms of documents on the second side, it equally means that these documents can be associated with many documents on the first side.

To reiterate an important modelling point about designing relationships, if you only consider one side of the relationship you may classify it as a “one” or as a “many” if you don’t consider both sides. The best advice is to ensure you ask the question of associativity from both directions and that you review your model a few times, especially for less apparent relationships.

There are several strategies but the 1-way embedding strategy optimises the read performance of a N:N relationship by embedding the references in one side of the relationship. The key step in this strategy is establishing the relationship balance and choosing the side which has one or two orders magnitude of difference in the number of entities. If the relationships are close to an even ratio then 2-way embedding is probably a better strategy.



Embed	Link
for integrity for read operations	when the "many" side is a huge number
for integrity for write operations on one-to-one and one-to-many	for integrity on write operations on many-to-many
for data that is deleted together	when a piece is frequently used, but not the other and memory is an issue
by default	

In order to help in the choice of relationship when you are modeling there are several helpful rules of thumb:

In terms of integrity, there are rules of thumb which related to when different pieces of information need to either be read or written together and where strong consistency is required.
For embedding data, this should be favoured for read operations to support integrity as well as for write operations on one-to-one and one-to-many relationships.
Embedding is also recommended when data is going to be deleted together.
In the majority of modeling designs, embedding data should be the default approach or choice taken.

In terms of linking data, it is recommended that this is done where there is a very large number of objects on the “many” side in either a one-to-many or a many-to-many relationship.
It is also recommended for write operations on many-to-many relationships.
Finally, linkin is recommended where only a subset of the data is frequently used whilst the rest of the data is not and where memory may be an issue. We’ll look at a concrete design pattern later in this lesson, called the subset pattern which relates to this concept.

Finally, the most important factor to keep to the forefront when you are modelling is the frequency of all the queries. This will help you make a better and more informed decision as to how you should model any specific data or relationship.

Quiz

Quiz

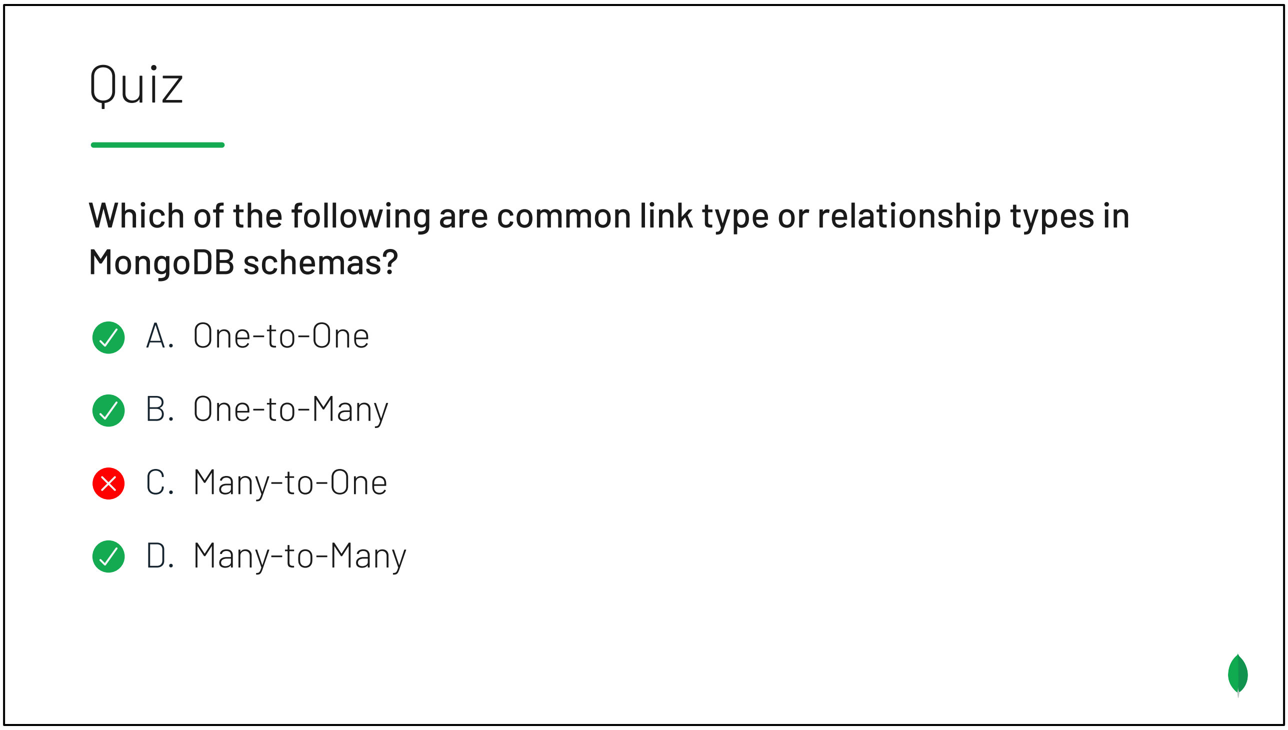
Which of the following are common link type or relationship types in MongoDB schemas? More than 1 answer choice can be correct.

● A. One-to-One

● B. One-to-Many

● C. Many-to-One

● D. Many-to-Many



Quiz

Which of the following are common link type or relationship types in MongoDB schemas?

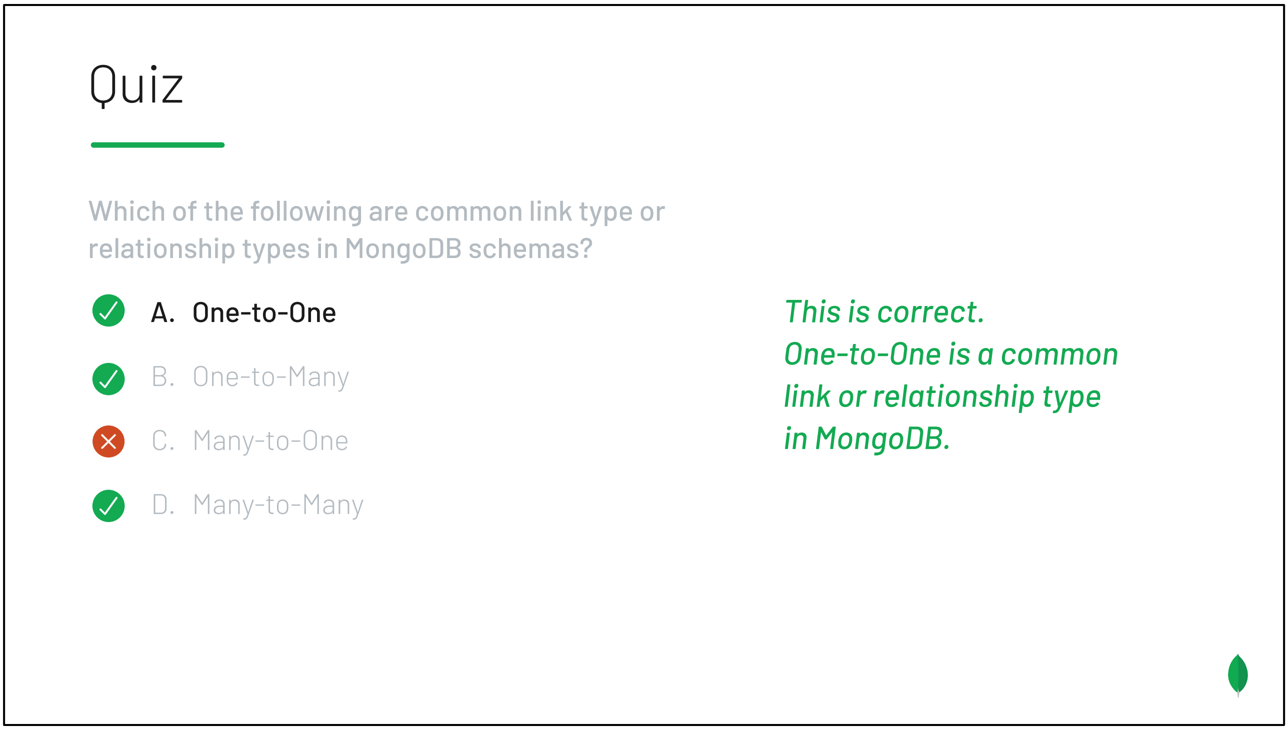
✓ A. One-to-One

✓ B. One-to-Many

✗ C. Many-to-One

✓ D. Many-to-Many

CORRECT: One-to-One, this is a common link or relationship type in MongoDB.
CORRECT: One-to-Many, this is a common link or relationship type in MongoDB.
INCORRECT: Many-to-One - This isn't a common relationship or link type in MongoDB.
CORRECT: Many-to-Many, this is a common link or relationship type in MongoDB. This is possible in MongoDB, an example would be where you have arrays on either side of this relationship.



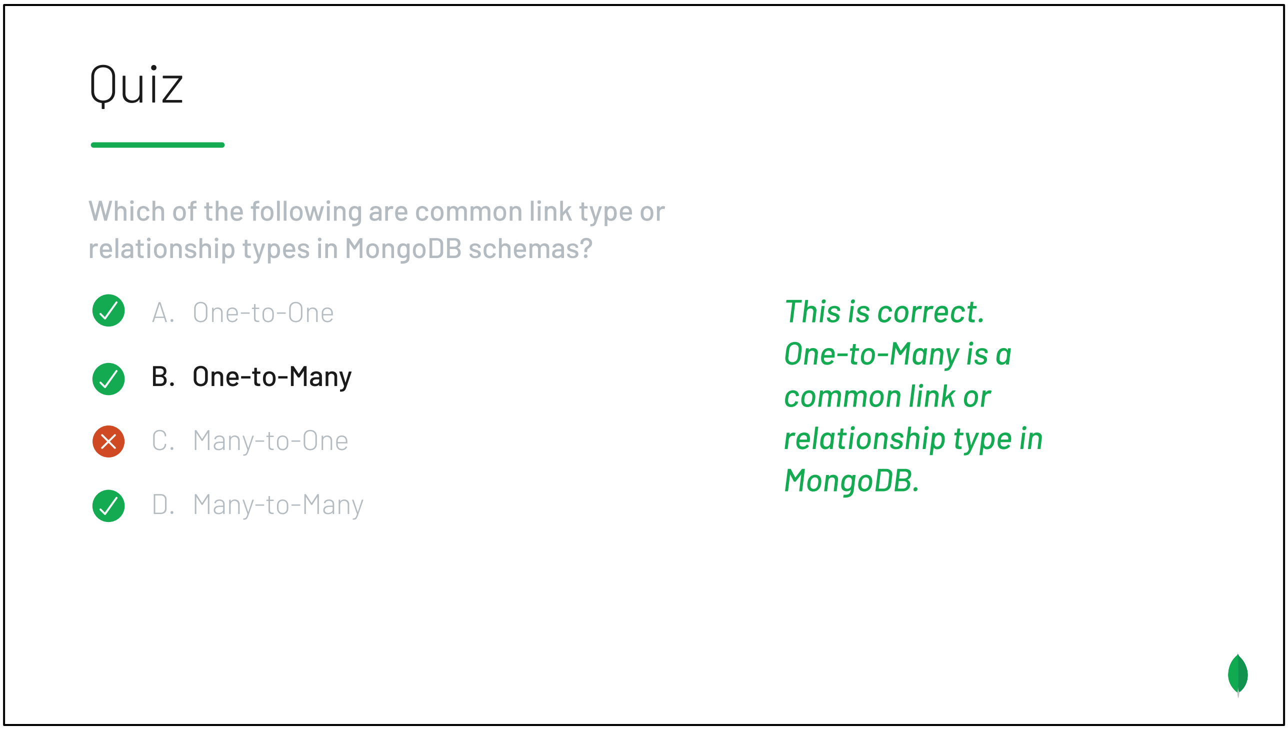
Quiz

Which of the following are common link type or relationship types in MongoDB schemas?

- ✓ A. One-to-One
- ✓ B. One-to-Many
- ✗ C. Many-to-One
- ✓ D. Many-to-Many

This is correct.
One-to-One is a common link or relationship type in MongoDB.

CORRECT: One-to-One - This is correct. One-to-One is a common link or relationship type in MongoDB.



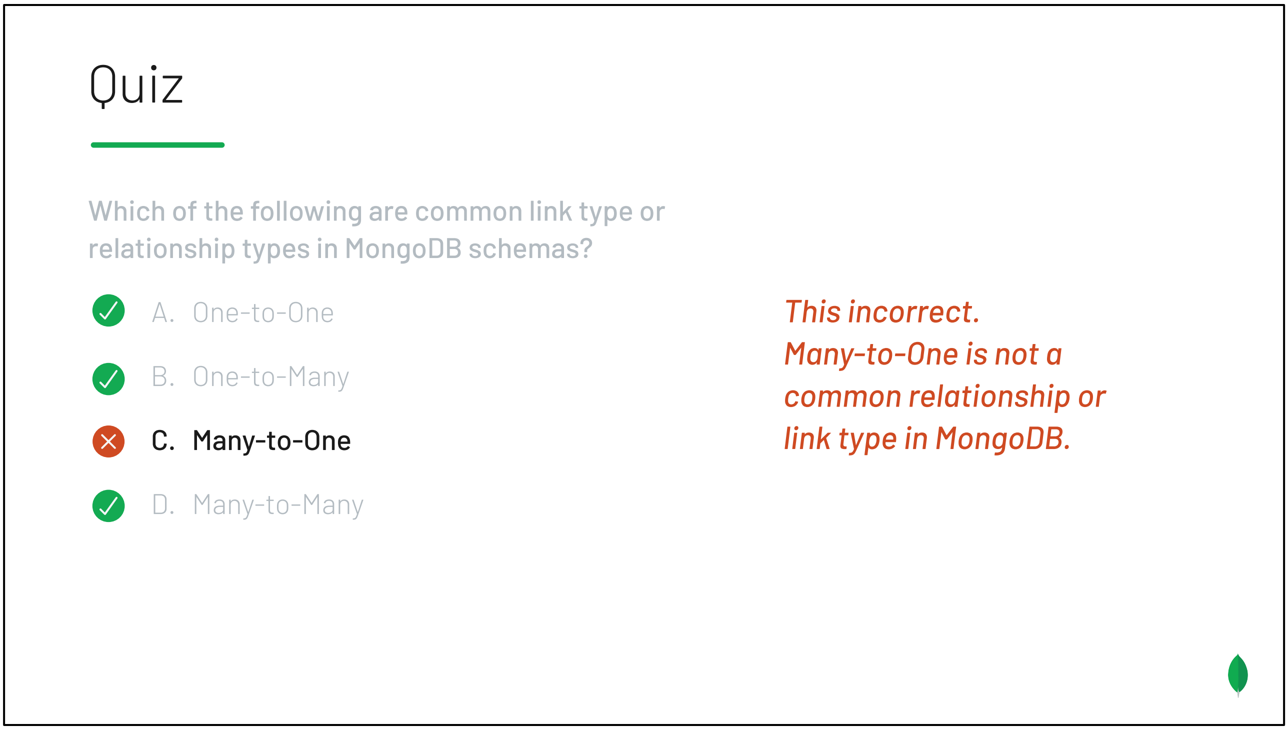
Quiz

Which of the following are common link type or relationship types in MongoDB schemas?

- ✓ A. One-to-One
- ✓ **B. One-to-Many**
- ✕ C. Many-to-One
- ✓ D. Many-to-Many

This is correct. One-to-Many is a common link or relationship type in MongoDB.

CORRECT: One-to-Many - This is correct. One-to-Many is a common link or relationship type in MongoDB.



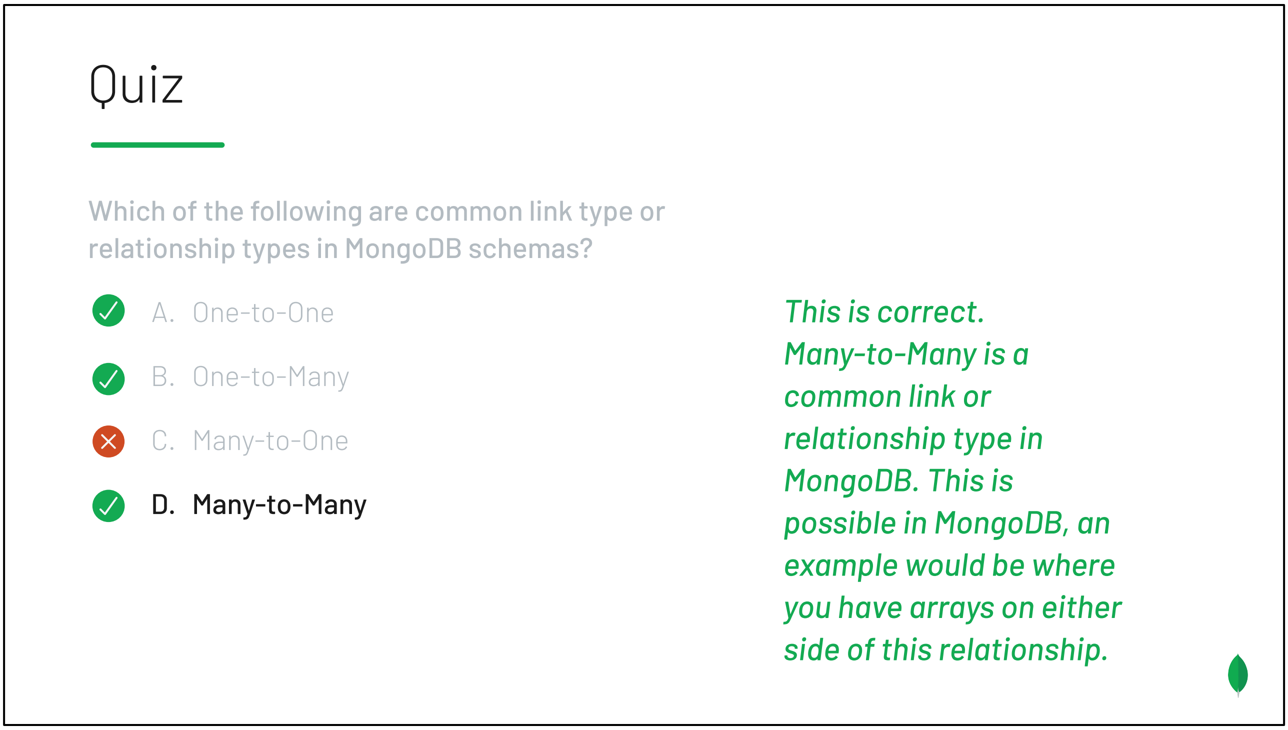
Quiz

Which of the following are common link type or relationship types in MongoDB schemas?

- ✓ A. One-to-One
- ✓ B. One-to-Many
- ✗ C. Many-to-One
- ✓ D. Many-to-Many

This incorrect. Many-to-One is not a common relationship or link type in MongoDB.

INCORRECT: Many-to-One - This is incorrect. Many-to-One is not a common relationship or link type in MongoDB.



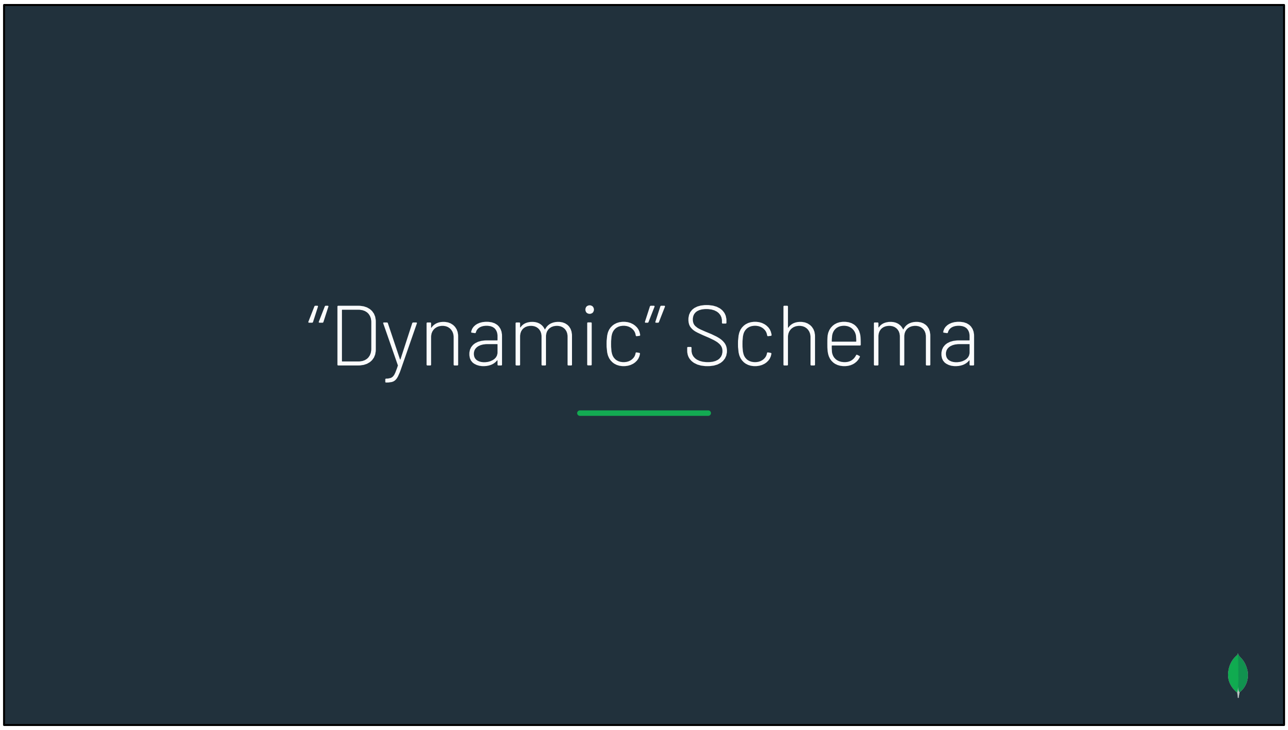
Quiz

Which of the following are common link type or relationship types in MongoDB schemas?

- ✓ A. One-to-One
- ✓ B. One-to-Many
- ✗ C. Many-to-One
- ✓ D. Many-to-Many

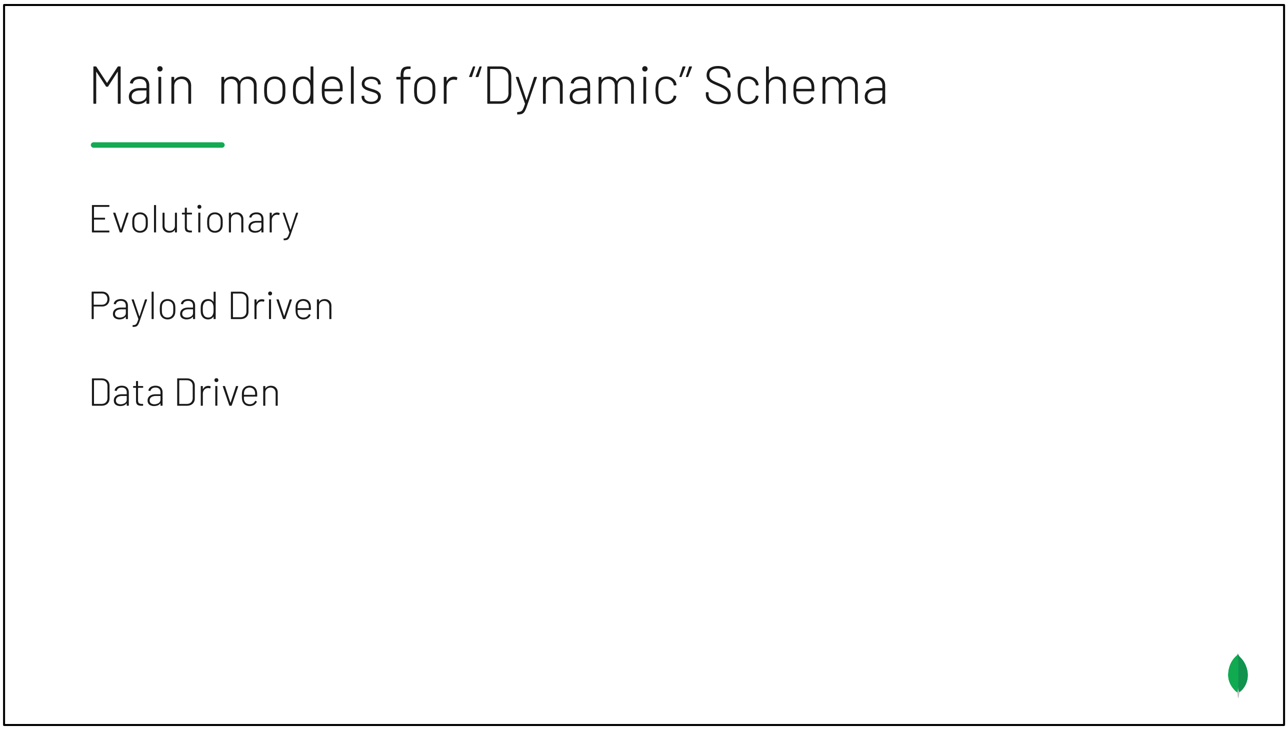
This is correct. Many-to-Many is a common link or relationship type in MongoDB. This is possible in MongoDB, an example would be where you have arrays on either side of this relationship.

CORRECT: Many-to-Many - This is correct. Many-to-Many is a common link or relationship type in MongoDB. This is possible in MongoDB, an example would be where you have arrays on either side of this relationship.



“Dynamic” Schema

Let’s introduce and define the various types of “Dynamic” schema.



Main models for “Dynamic” Schema

Evolutionary

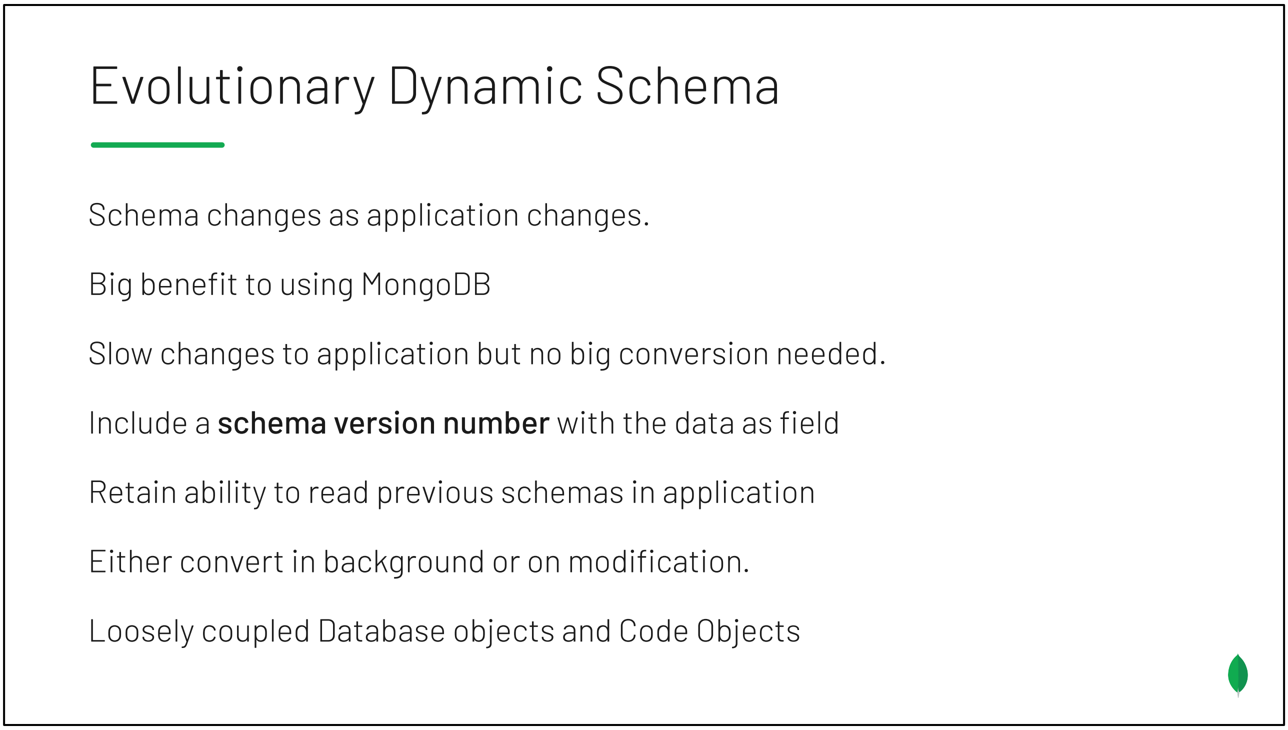
Payload Driven

Data Driven

There are three main models of dynamic schema, we’ll explore each in a sequence.

Evolutionary is where the schema changes as your application changes.
Payload driven is where your application receives data from another application and has very little control over the schema.
Data driven is where the field names actually represent data.

We’ll look at each of these three models in depth now.



Evolutionary Dynamic Schema

Schema changes as application changes.

Big benefit to using MongoDB

Slow changes to application but no big conversion needed.

Include a **schema version number** with the data as field

Retain ability to read previous schemas in application

Either convert in background or on modification.

Loosely coupled Database objects and Code Objects

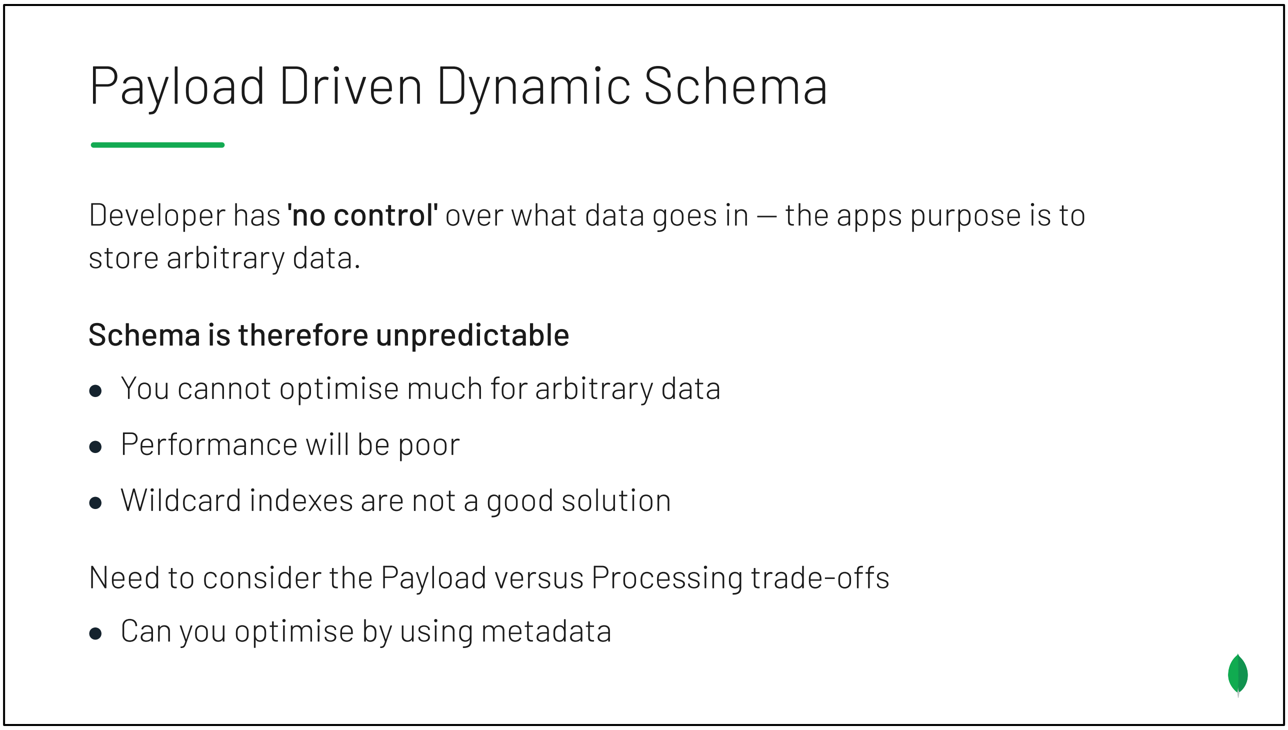
Schema versioning can be used to achieve a dynamic schema that changes with your application. We will also cover this topic in the patterns section later in this lesson.

This type of approach can provide significant benefits when using MongoDB as changes to your code can be easily accommodated by your database.

It further allows for easy backward compatibility through the use of a schema version number so older schemas can easily be read and processed without significant application changes.

This type of schema means that changes can be modified when there is a document modification or in the background, there is no need for a complete update of all documents to the latest version in one bulk converison.

This allows for loosely couple database objects and code objects.



Payload Driven Dynamic Schema

Developer has **'no control'** over what data goes in — the apps purpose is to store arbitrary data.

Schema is therefore unpredictable

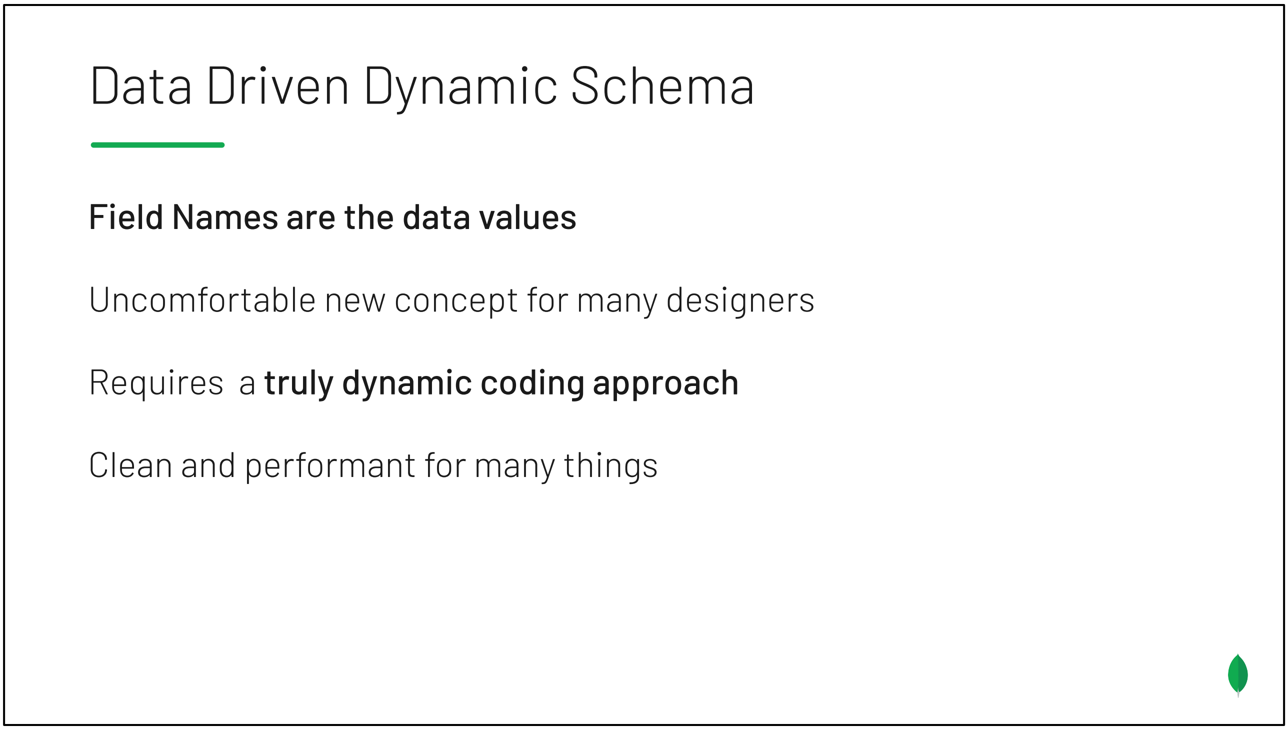
- You cannot optimise much for arbitrary data
- Performance will be poor
- Wildcard indexes are not a good solution

Need to consider the Payload versus Processing trade-offs

- Can you optimise by using metadata

In some applications, the developer is merely passed data of which there is no possibility to change the schema and they must store it as is.
It is also likely that the schema could change and due to this unpredictability, performance will be not be possible.
Some optimisation can be achieve through using metadata about the data but the core data sent to the application itself isn't optimisible.

In these situations, it is difficult to optimise and the trade-offs need to be consider in terms of payload and of processing.



Data Driven Dynamic Schema

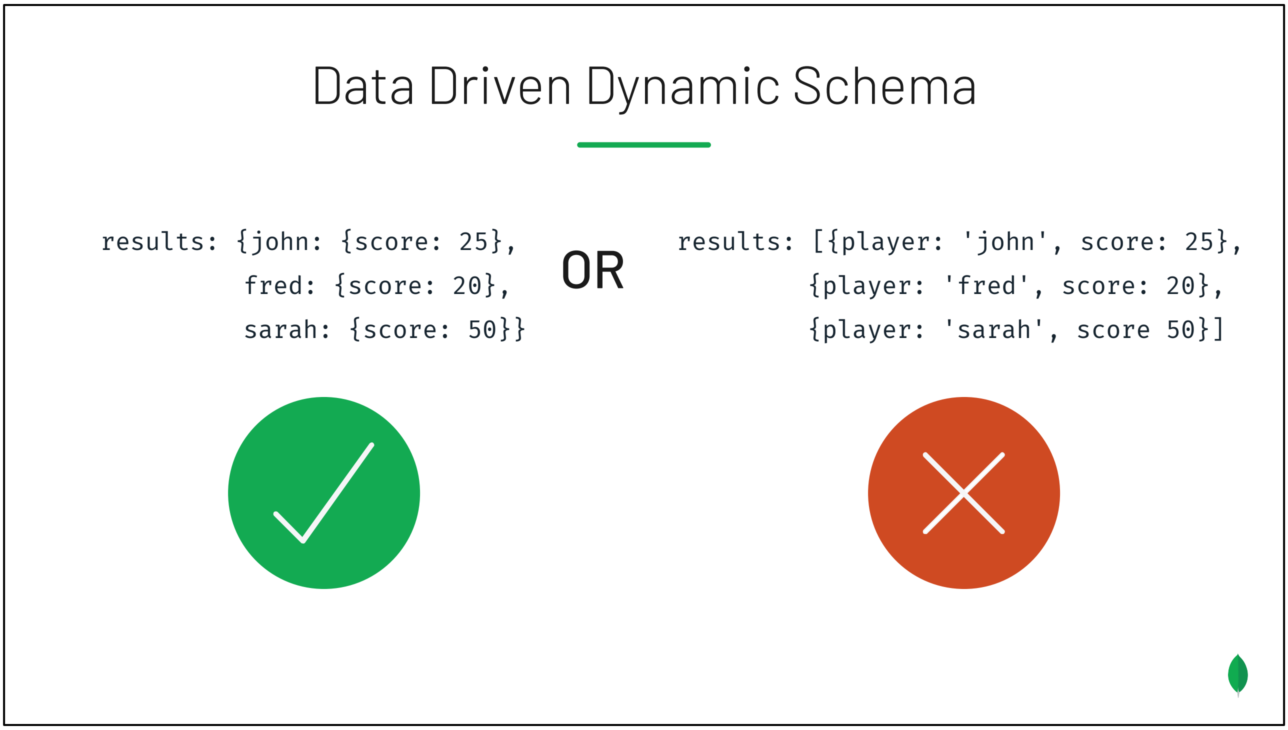
Field Names are the data values

Uncomfortable new concept for many designers

Requires a **truly dynamic coding approach**

Clean and performant for many things

Let’s look at the data driven dynamic schema approach, it can be highly performant but does require a different coding approach.
Instead of using a field name, the field name itself is representing data. This does require a different approach to coding but it is performant and clean when compared to other approaches.



Data Driven Dynamic Schema

```
results: {john: {score: 25},
          fred: {score: 20},
          sarah: {score: 50}}
```

✓

OR

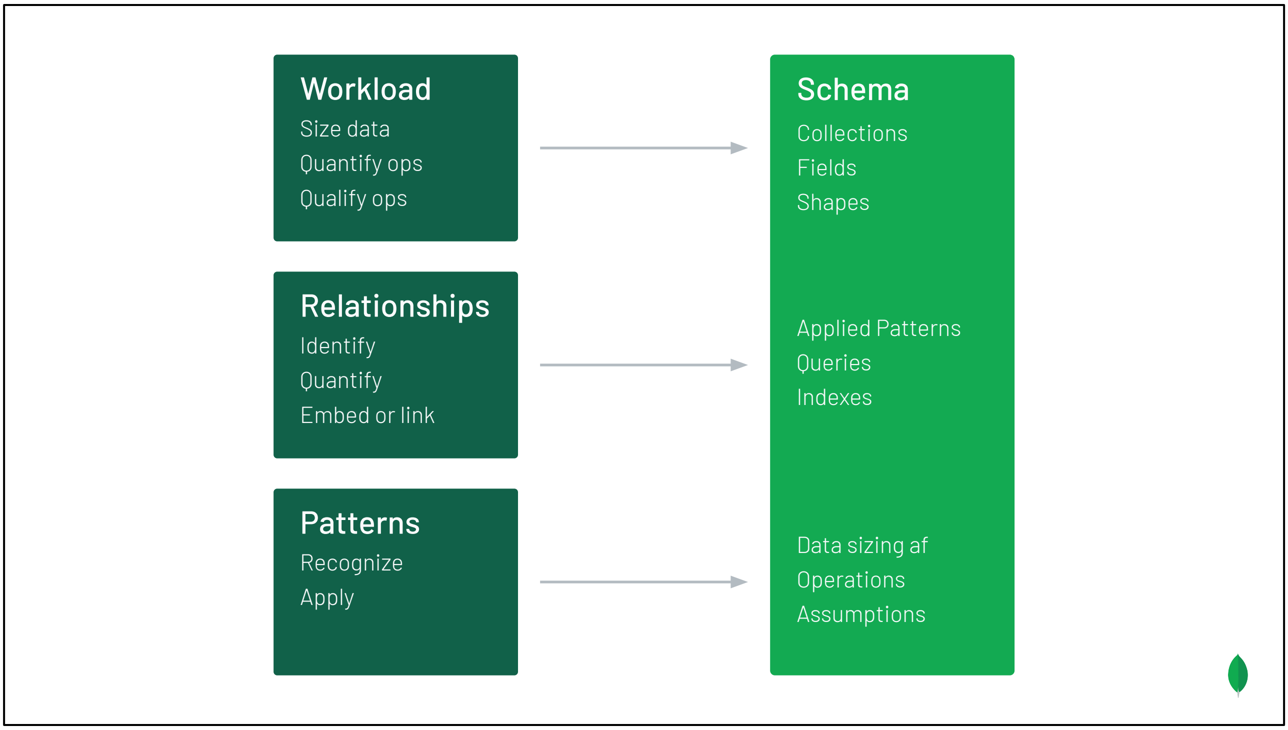
```
results: [{player: 'john', score: 25},
          {player: 'fred', score: 20},
          {player: 'sarah', score 50}]
```

✗

Which of the two examples use a data driven dynamic schema ? Recall that in a data driven dynamic schema, the field itself typically represents data.

Modelling Methodology

Let’s look at the methodology we recommend that you follow when modelling your schema.



Workload
Size data
Quantify ops
Qualify ops

Relationships
Identify
Quantify
Embed or link

Patterns
Recognize
Apply

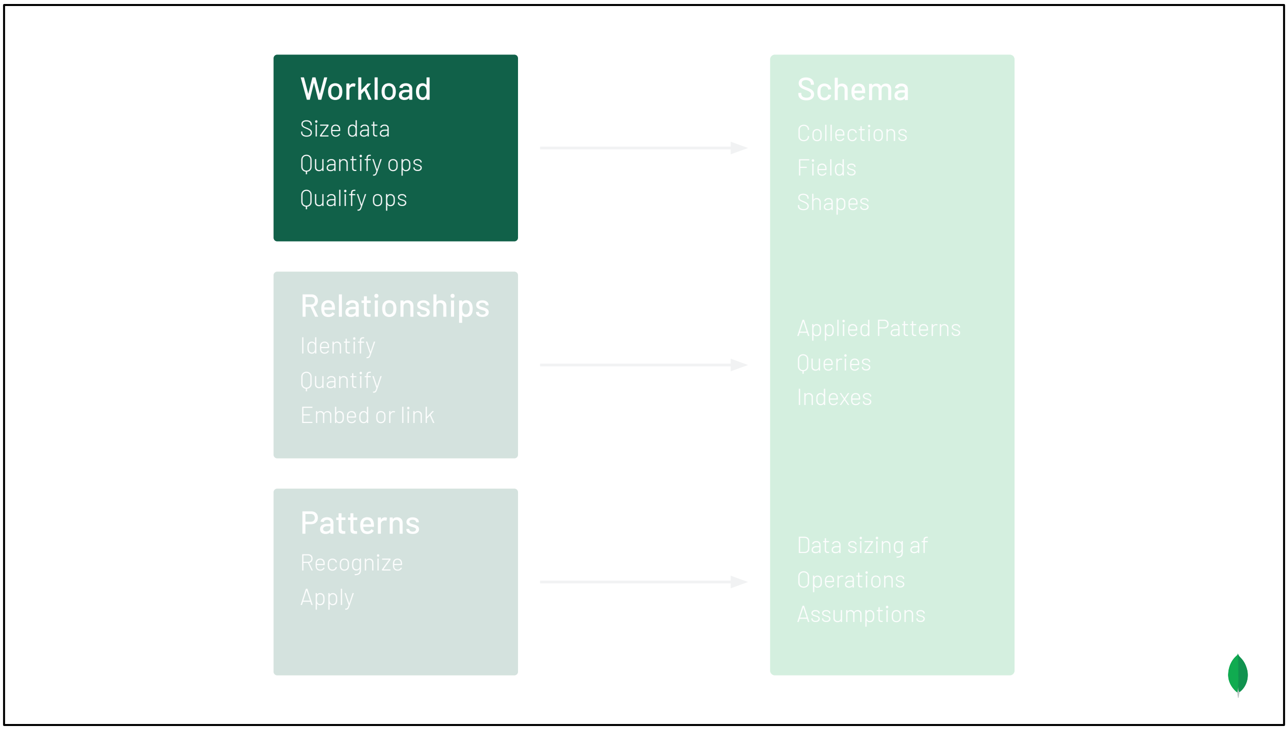
Schema
Collections
Fields
Shapes

Applied Patterns
Queries
Indexes

Data sizing af
Operations
Assumptions

The first stage of the modelling methodology we recommend for MongoDB is to classify the workload.

This classification involves calculating the size of the data as well identify the important reads and writes. You can identify the important operations only after determining how many of each (reads/writes) will occur and which each will be (a read or a write).



Workload

Size data

Quantify ops

Qualify ops

Relationships

Identify

Quantify

Embed or link

Patterns

Recognize

Apply

Schema

Collections

Fields

Shapes

Applied Patterns

Queries

Indexes

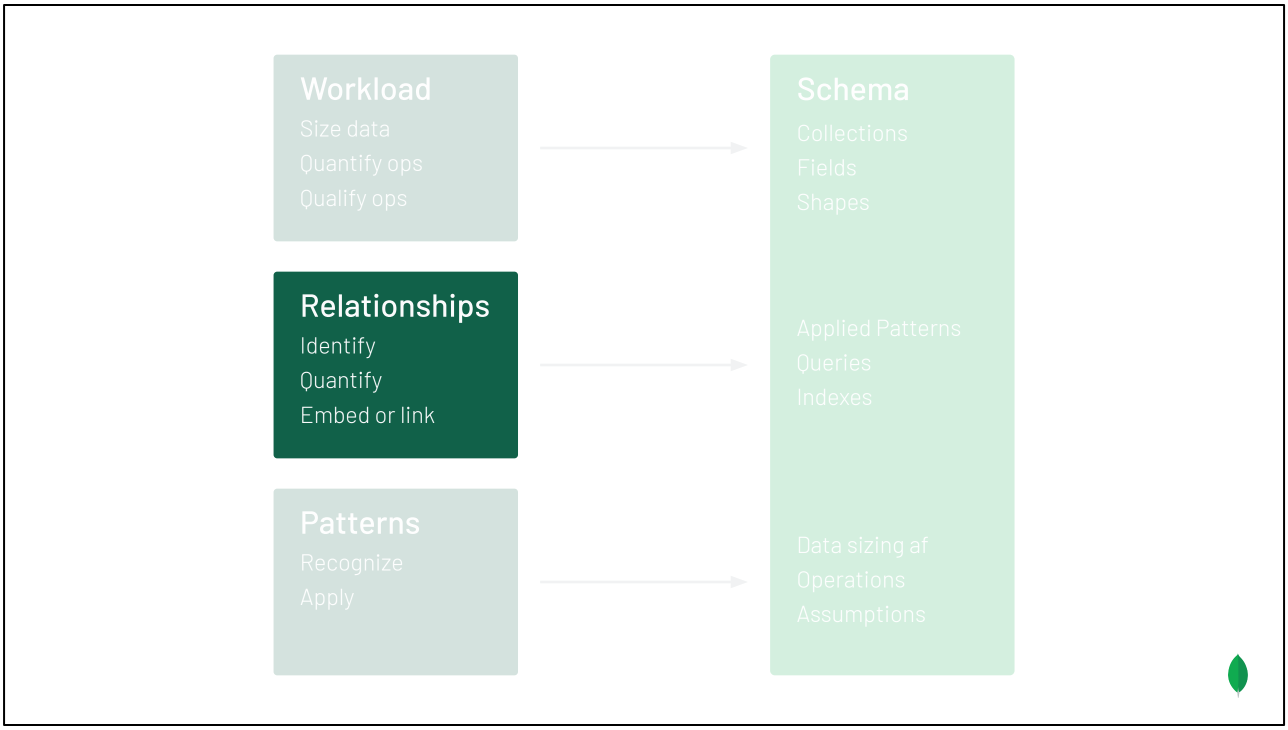
Data sizing af

Operations

Assumptions

The first stage of the modelling methodology we recommend for MongoDB is to classify the workload.

This classification involves calculating the size of the data as well identify the important reads and writes. You can identify the important operations only after determining how many of each (reads/writes) will occur and which each will be (a read or a write).



Workload
Size data
Quantify ops
Qualify ops

Relationships
Identify
Quantify
Embed or link

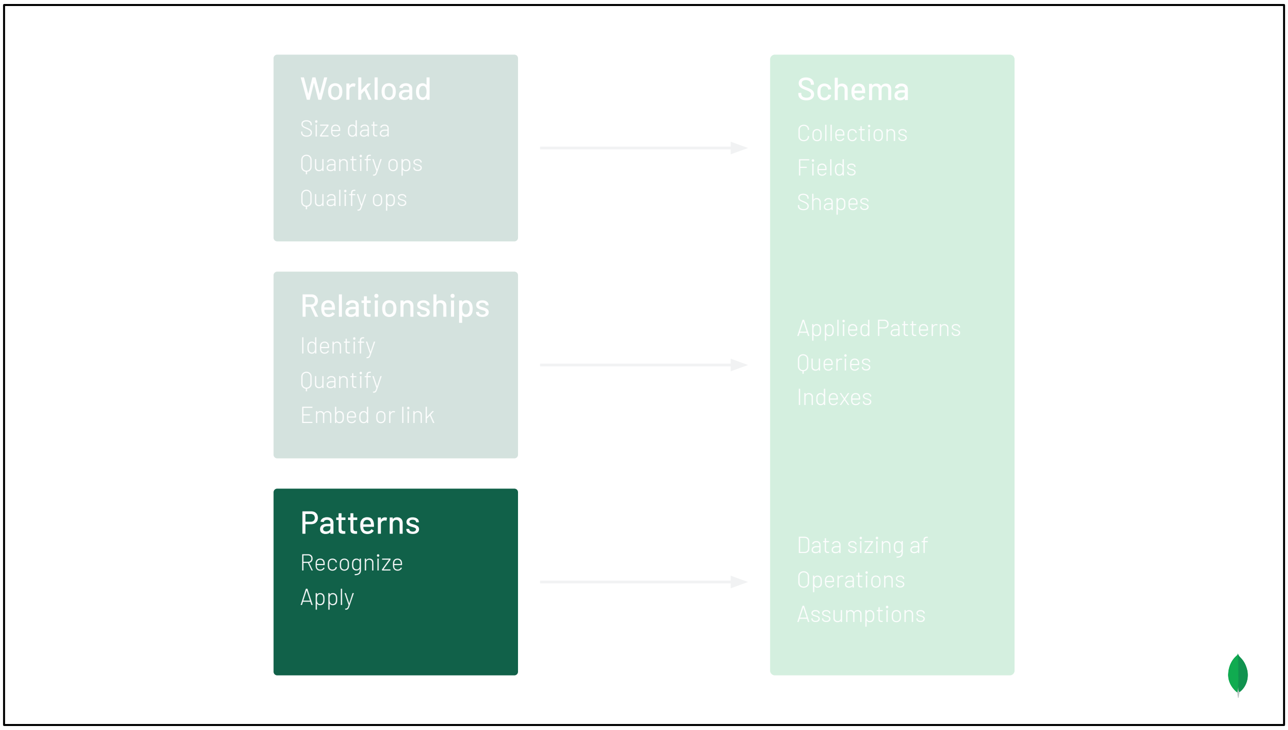
Patterns
Recognize
Apply

Schema
Collections
Fields
Shapes

Applied Patterns
Queries
Indexes

Data sizing af
Operations
Assumptions

The second stage of the modelling methodology we recommend for MongoDB is to classify the relationships in the data.
This process involves identifying all the relationships between the data and then how to link or embed the related entities.



Workload

Size data

Quantify ops

Qualify ops

Relationships

Identify

Quantify

Embed or link

Patterns

Recognize

Apply

Schema

Collections

Fields

Shapes

Applied Patterns

Queries

Indexes

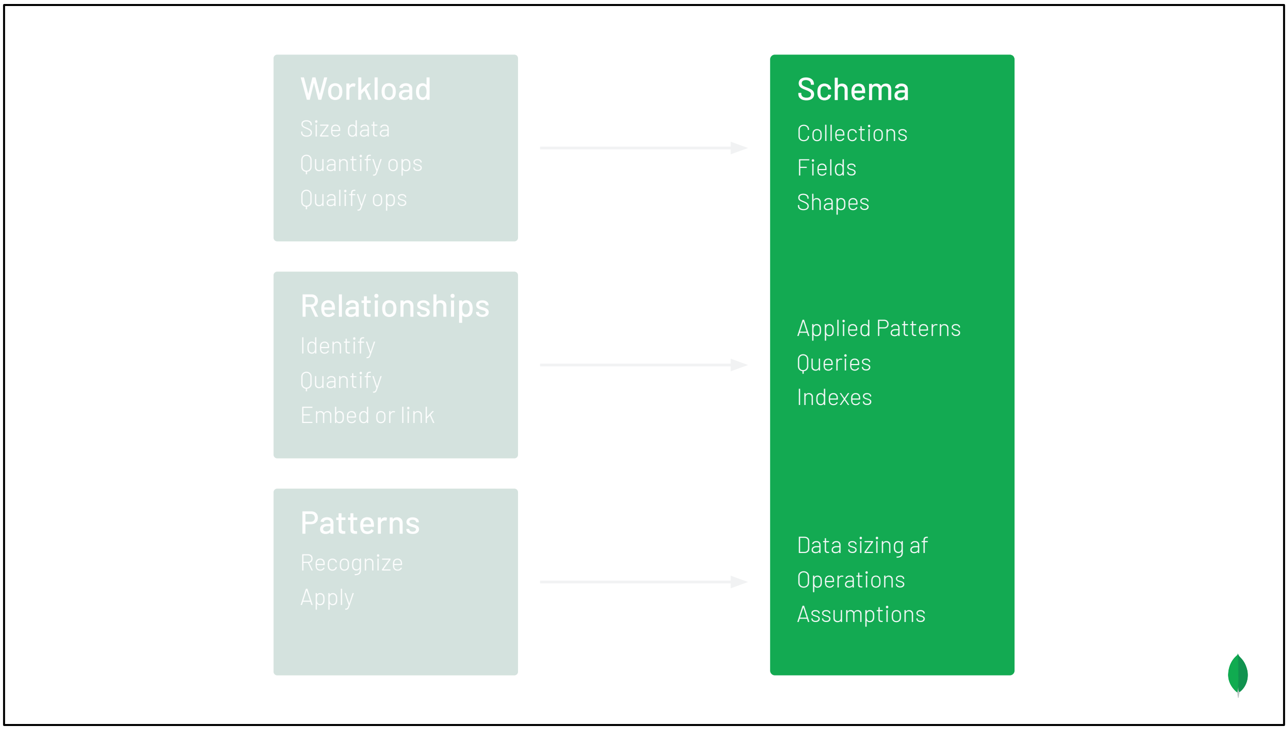
Data sizing af

Operations

Assumptions

The third stage of the modelling methodology we recommend for MongoDB is to identify if there are opportunities to use a pattern or patterns with the schema design. If there is are opportunities, this phase includes apply these and refactoring the existing schema to use the appropriate pattern or patterns.

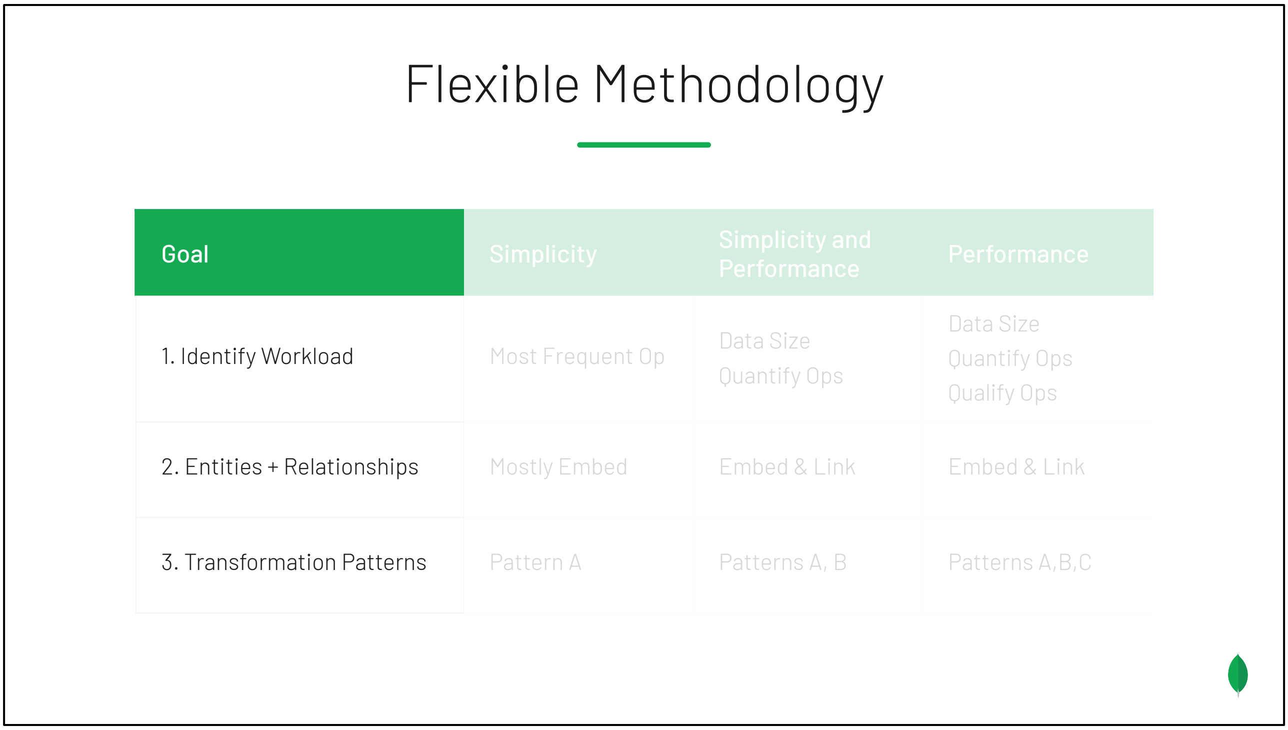
Patterns are useful, however a good rule of thumb is only to apply patterns that are needed for optimizations. Patterns can add functionality to the specific application’s code so the best advice is to use them liberally and where there are clear gains.



After modelling your application and it's workload, relationships, and patterns which might apply you can then generate the most suitable schema.

This includes determine how many collection, what fields are needed in documents, what queries are required and what indexes should be provided to support those queries.

This along with the data size and expected data growth as well as the mix of operation types and any other assumptions should be documented to ensure you have a clearly defined schema that you can easily implement in MongoDB.



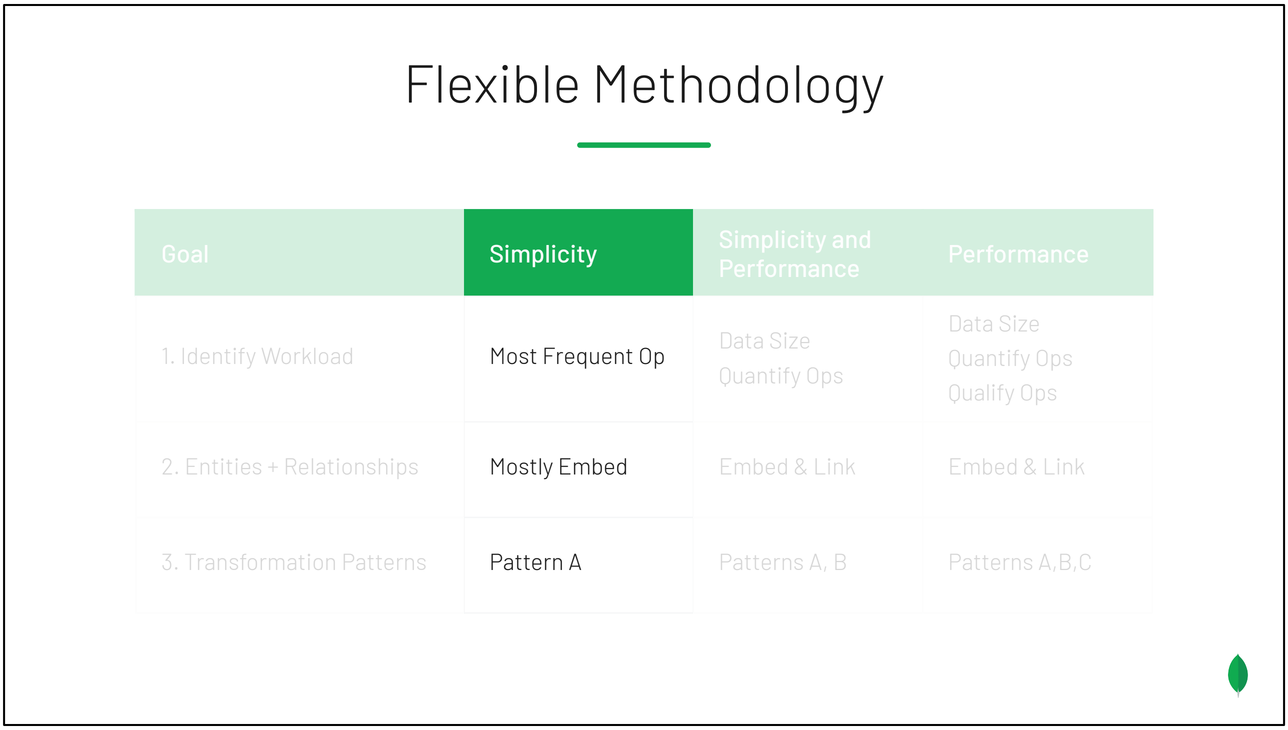
Flexible Methodology

Goal	Simplicity	Simplicity and Performance	Performance
1. Identify Workload	Most Frequent Op	Data Size Quantify Ops	Data Size Quantify Ops Qualify Ops
2. Entities + Relationships	Mostly Embed	Embed & Link	Embed & Link
3. Transformation Patterns	Pattern A	Patterns A, B	Patterns A,B,C

We propose a flexible methodology, in that you don't need to apply all the steps all the time, and you should use it as and when needed. It should be seen as optional rather than prescriptive.

As we've seen in the previous slides, the three steps are to identify the workload, to identify the relationships and entities, and to determine if any schema design patterns could apply. Let's look at these steps from three different developer goals.

The goals are to have the simplest possible schema, or to have a simple but yet performant schema, or to focus on having a highly performant schema. Developers can focus on one of these goals and we'll talk in the next slides about how they can use this modeling methodology to satisfy any one of these goals.



Flexible Methodology

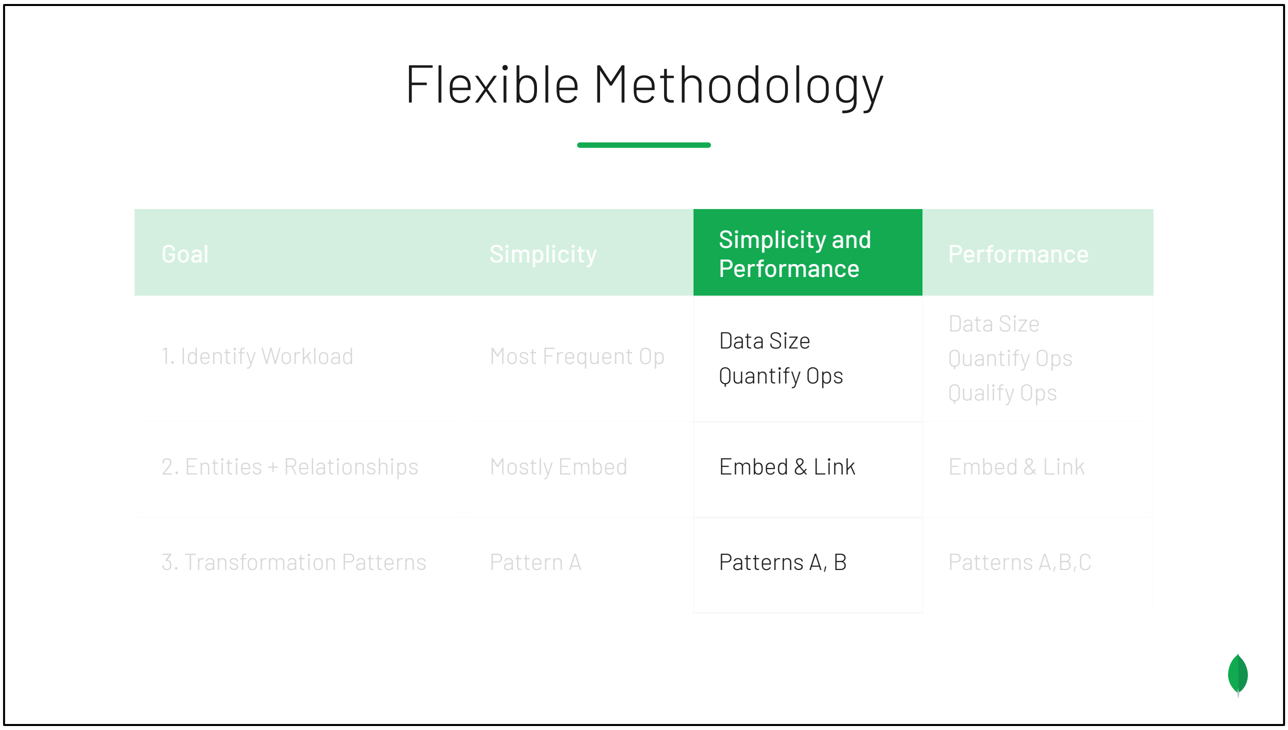
Goal	Simplicity	Simplicity and Performance	Performance
1. Identify Workload	Most Frequent Op	Data Size Quantify Ops	Data Size Quantify Ops Qualify Ops
2. Entities + Relationships	Mostly Embed	Embed & Link	Embed & Link
3. Transformation Patterns	Pattern A	Patterns A, B	Patterns A,B,C

If your goal is simplicity, you can focus on these specific aspects.

For the workload, you should identify the most frequent operation.

For the relationships, you should try and embedded these rather than linking them.

For any patterns, you should try and use one or none to keep it simple.



Flexible Methodology

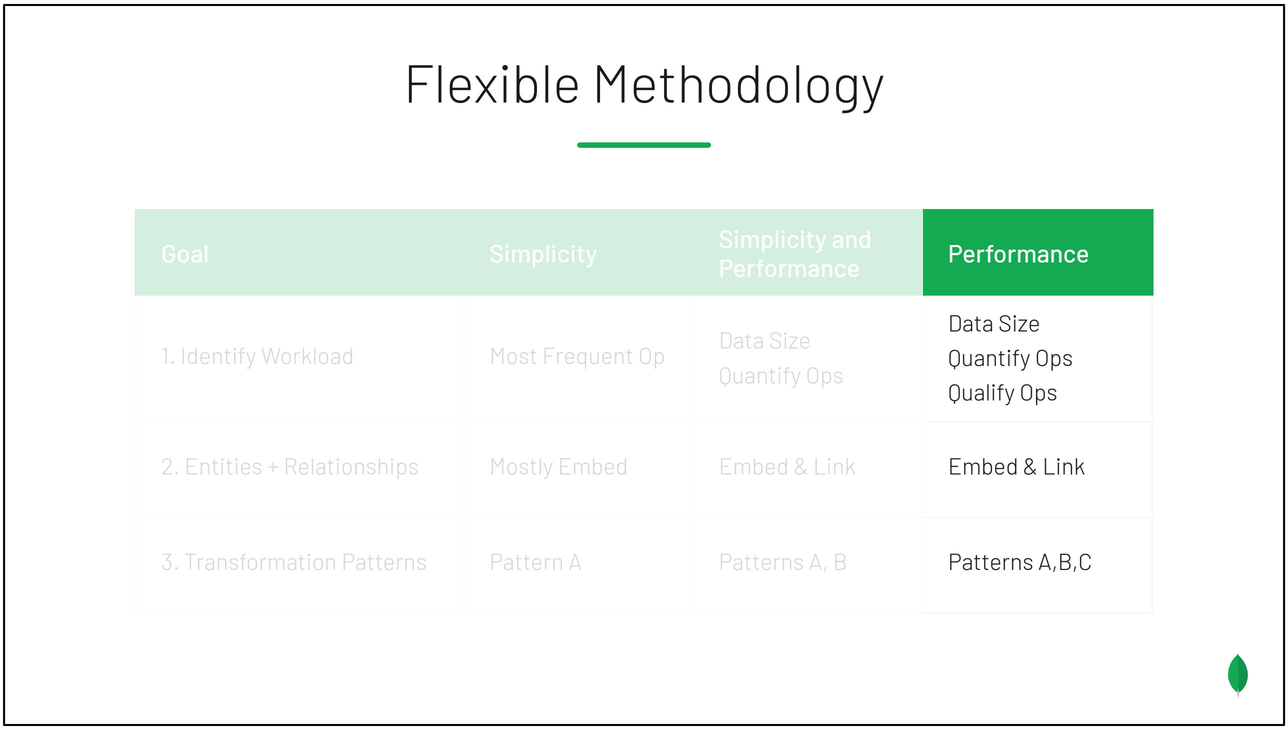
Goal	Simplicity	Simplicity and Performance	Performance
1. Identify Workload	Most Frequent Op	Data Size Quantify Ops	Data Size Quantify Ops Qualify Ops
2. Entities + Relationships	Mostly Embed	Embed & Link	Embed & Link
3. Transformation Patterns	Pattern A	Patterns A, B	Patterns A,B,C

If you want to balance some performance with simplicity, you can focus on these specific aspects.

For the workload, you should identify the data size and quantify the various operations.

For the relationships, you should try embedded and link as your data.

For any patterns, you should use one or two patterns at most.



Flexible Methodology

Goal	Simplicity	Simplicity and Performance	Performance
1. Identify Workload	Most Frequent Op	Data Size Quantify Ops	Data Size Quantify Ops Qualify Ops
2. Entities + Relationships	Mostly Embed	Embed & Link	Embed & Link
3. Transformation Patterns	Pattern A	Patterns A, B	Patterns A,B,C

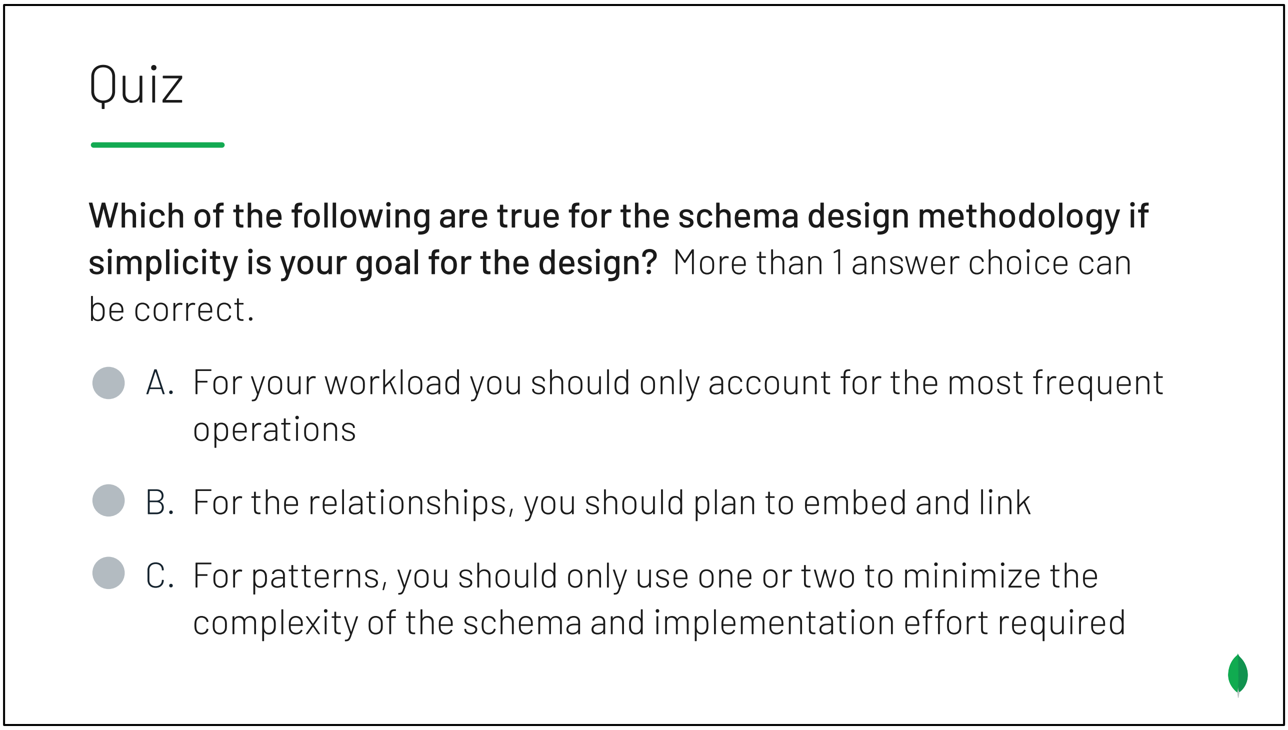
If you want to focus on performance, you can focus on these specific aspects. Focusing on performance will likely increase the complexity of the application and will make it more difficult to maintain.

For the workload, you should identify the data size, quantify the various operations and also qualify those operations.

For the relationships, you should try embedded and link as your data.

For any patterns, you should use as many applicable patterns (two or more) that provide performance advantages.

Quiz



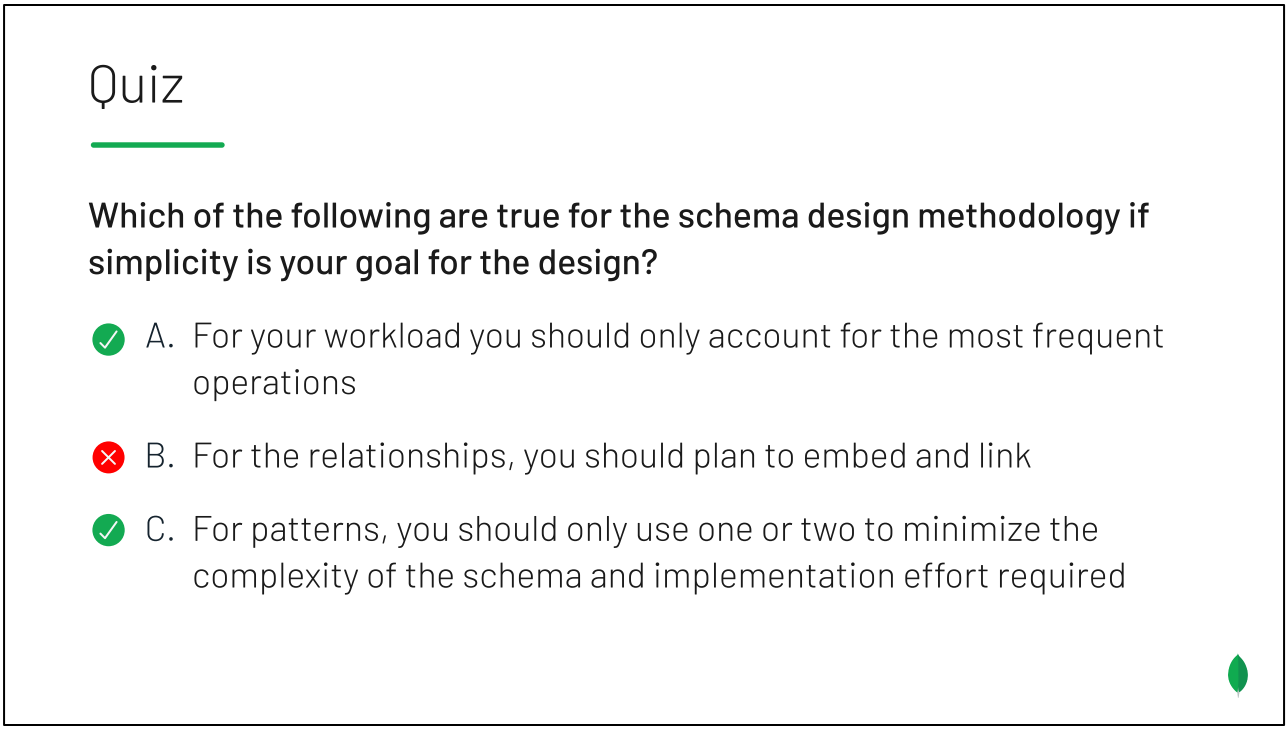
Quiz

Which of the following are true for the schema design methodology if simplicity is your goal for the design? More than 1 answer choice can be correct.

● A. For your workload you should only account for the most frequent operations

● B. For the relationships, you should plan to embed and link

● C. For patterns, you should only use one or two to minimize the complexity of the schema and implementation effort required



Quiz

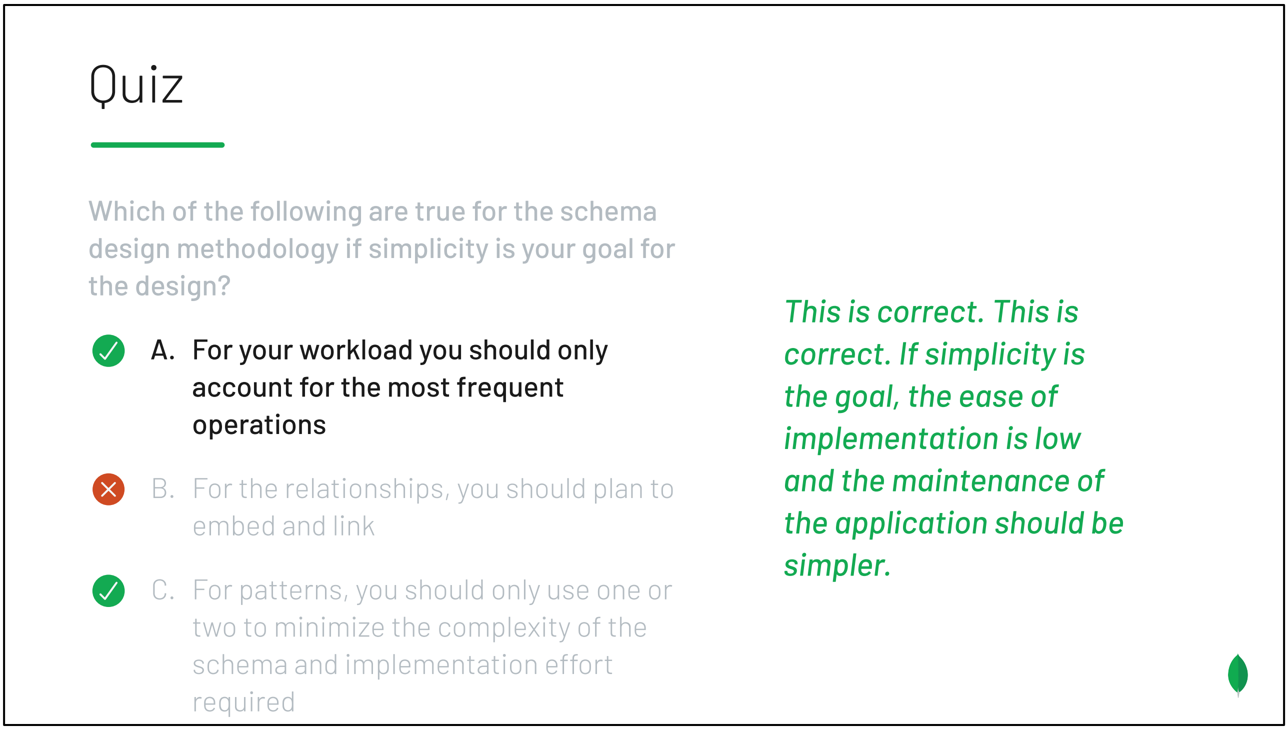
Which of the following are true for the schema design methodology if simplicity is your goal for the design?

✓ A. For your workload you should only account for the most frequent operations

✗ B. For the relationships, you should plan to embed and link

✓ C. For patterns, you should only use one or two to minimize the complexity of the schema and implementation effort required

CORRECT: For your workload you should only account for the most frequent operations - If simplicity is the goal, the ease of implementation is low and the maintenance of the application should be simpler
INCORRECT: For the relationships, you should plan to embed and link - Incorrect, in the context of simplicity you should only embed relationships
CORRECT: For patterns, you should only use one or two to minimize the complexity of the schema and implementation effort required - A minimum number of patterns should be chosen when the focus is simplicity



Quiz

Which of the following are true for the schema design methodology if simplicity is your goal for the design?

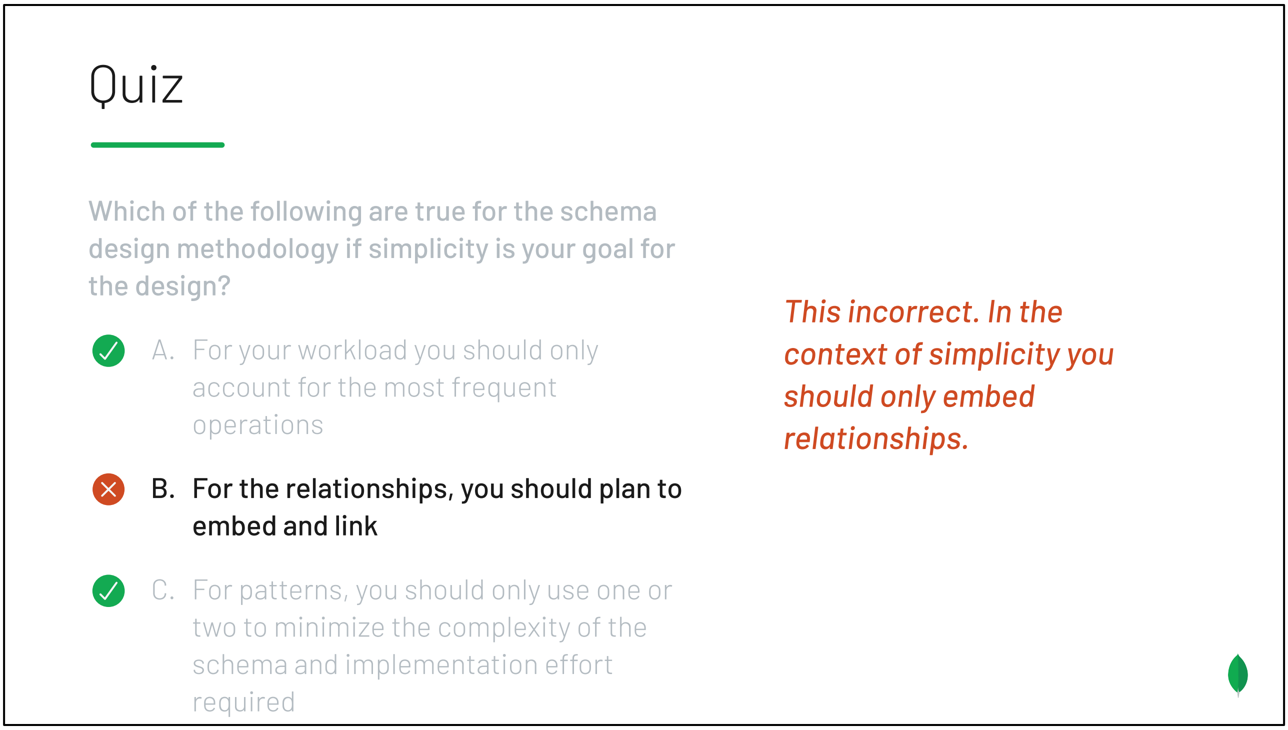
✓ A. For your workload you should only account for the most frequent operations

✗ B. For the relationships, you should plan to embed and link

✓ C. For patterns, you should only use one or two to minimize the complexity of the schema and implementation effort required

This is correct. This is correct. If simplicity is the goal, the ease of implementation is low and the maintenance of the application should be simpler.

CORRECT: For your workload you should only account for the most frequent operations - This is correct. If simplicity is the goal, the ease of implementation is low and the maintenance of the application should be simpler.



Quiz

Which of the following are true for the schema design methodology if simplicity is your goal for the design?

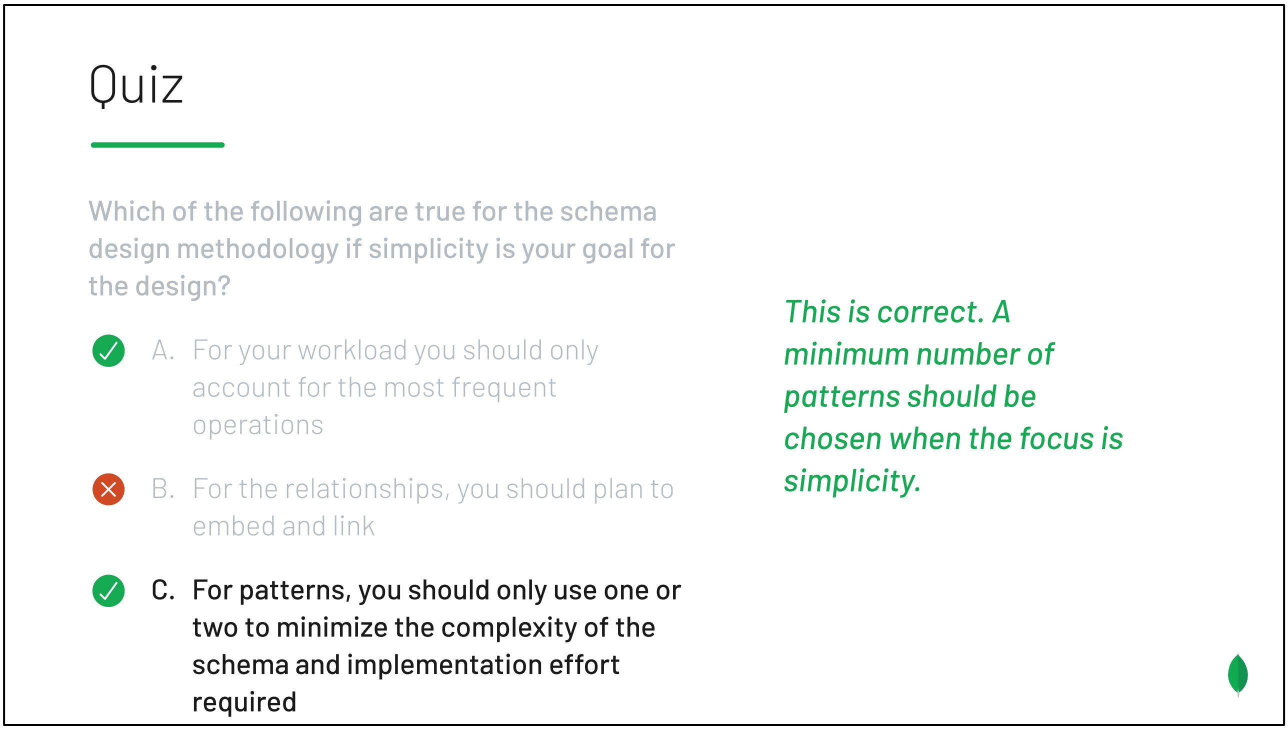
✓ A. For your workload you should only account for the most frequent operations

✗ B. For the relationships, you should plan to embed and link

✓ C. For patterns, you should only use one or two to minimize the complexity of the schema and implementation effort required

This incorrect. In the context of simplicity you should only embed relationships.

INCORRECT: For the relationships, you should plan to embed and link - This is incorrect. In the context of simplicity you should only embed relationships



Quiz

Which of the following are true for the schema design methodology if simplicity is your goal for the design?

✓ A. For your workload you should only account for the most frequent operations

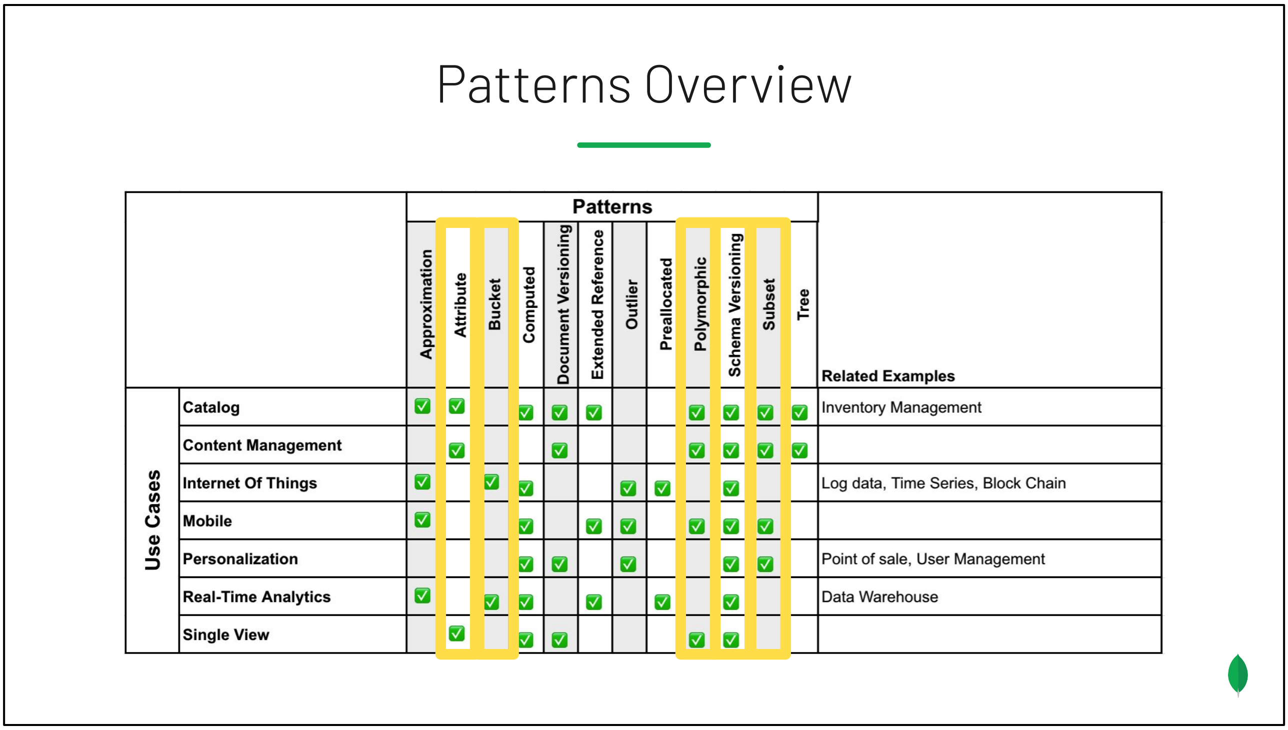
✗ B. For the relationships, you should plan to embed and link

✓ C. **For patterns, you should only use one or two to minimize the complexity of the schema and implementation effort required**

This is correct. A minimum number of patterns should be chosen when the focus is simplicity.

CORRECT: For patterns, you should only use one or two to minimize the complexity of the schema and implementation effort required - This is correct. A minimum number of patterns should be chosen when the focus is simplicity

Patterns Overview



Patterns Overview

		Patterns												
		Approximation	Attribute	Bucket	Computed	Document Versioning	Extended Reference	Outlier	Preallocated	Polymorphic	Schema Versioning	Subset	Tree	Related Examples
Use Cases	Catalog	✓	✓		✓	✓	✓			✓	✓	✓	✓	Inventory Management
	Content Management		✓			✓				✓	✓	✓	✓	
	Internet Of Things	✓		✓	✓			✓	✓		✓			Log data, Time Series, Block Chain
	Mobile	✓			✓		✓	✓		✓	✓	✓		
	Personalization				✓	✓		✓			✓	✓		Point of sale, User Management
	Real-Time Analytics	✓		✓	✓		✓		✓		✓			Data Warehouse
	Single View		✓		✓	✓				✓	✓			

There are a range of patterns that can be used to support certain use cases. We'll look at a number of these patterns to give some examples of how they are useful and when you might use them.

Firstly, we'll look at the Polymorphic pattern, which is useful when there are a variety of documents that have more similarities than differences and the documents need to be kept in a single collection.

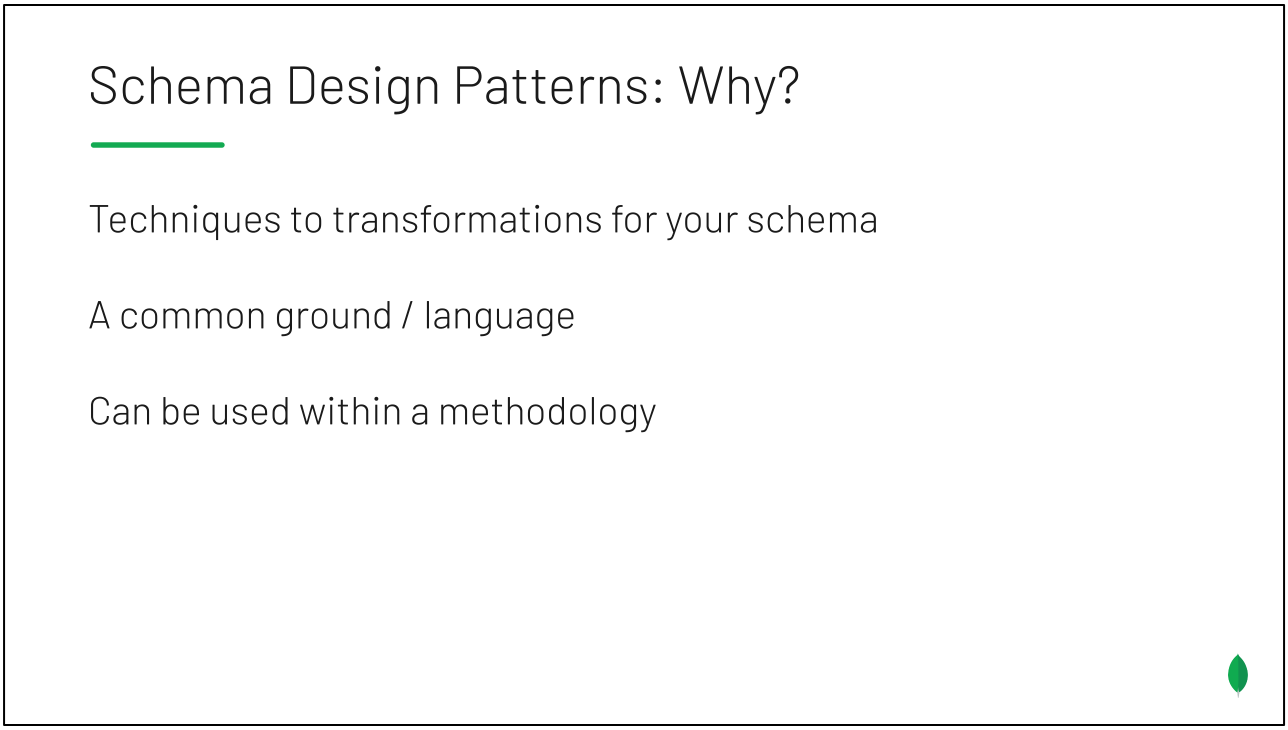
We'll then look at the Schema Versioning pattern, which allows for previous and current versions of documents to exist side by side in a collection.

Then we'll look at the Attribute patten, where there are big documents in a collection but we are only interested in a subset of fields that share common characteristics and we want to sort or query on that subset of fields.

After the Attribute pattern, we'll investigate the Bucket pattern, is useful for when we need to manage streaming data such as those from sensors/Internet of Things devices but it can also be used in other areas such as financial services. We'll show examples of both.

Finally, we'll look at the Subset pattern, helps resolve when your working set exceeds the capacity of RAM due to large documents and where a large amount of the data in the document isn't being used by the main/frequent queries to your database.

When I send the slide deck out after class, you’ll see a link to Patterns Summary blog https://www.mongodb.com/blog/post/building-with-patterns-a-summary. Feel free to visit that link to learn more about the various schema patterns.



Schema Design Patterns: Why?

Techniques to transformations for your schema

A common ground / language

Can be used within a methodology

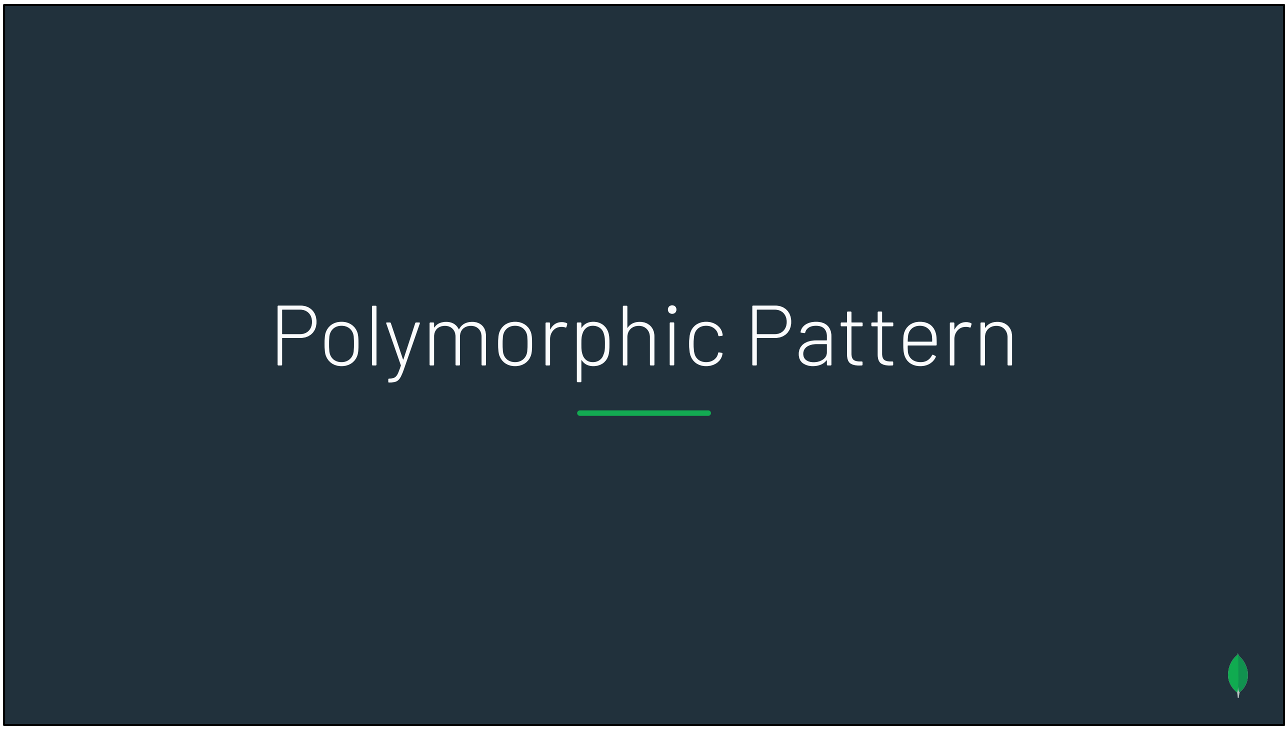
Document databases are different to relational database and many of the approaches used in relational data modelling don't apply. This often leads people to ask questions like "I'm creating an application to do thing X, how best should I model the data in MongoDB to support this?".

Patterns are guidance to try and help answer these types of modeling questions.

Patterns allow you to easily transform your schema to better support the specific use case.

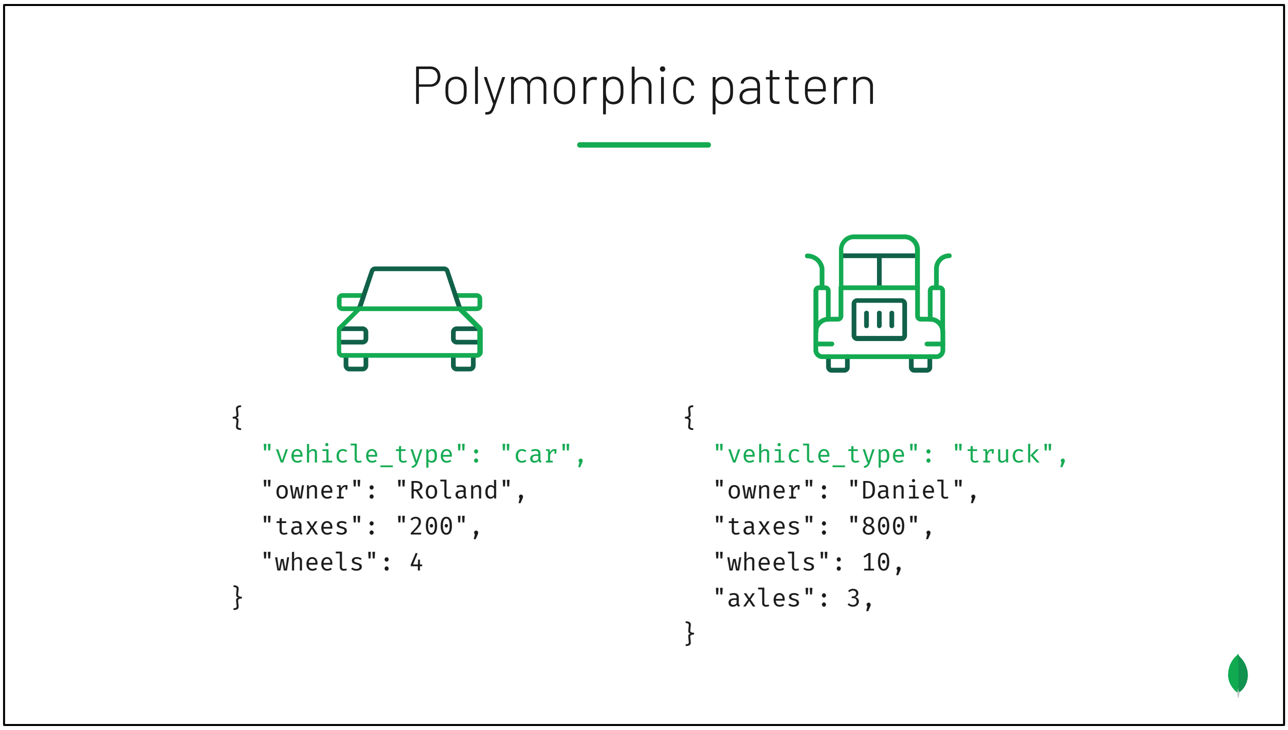
Patterns are designed to use a common set of terms and indeed the patterns themselves to help ensure everyone understands what is being discussed.

Patterns can be used alone or as we suggest within a methodology. We highlighted our recommended methodology earlier in the lesson and this has explicitly designed to include using patterns.



Polymorphic Pattern

Let’s look at our first pattern, the Polymorphic Pattern. This models situations where the date being modelled is more similar than different and we want to keep the data in the same collection.



Polymorphic pattern

```
{
  "vehicle_type": "car",
  "owner": "Roland",
  "taxes": "200",
  "wheels": 4
}
```

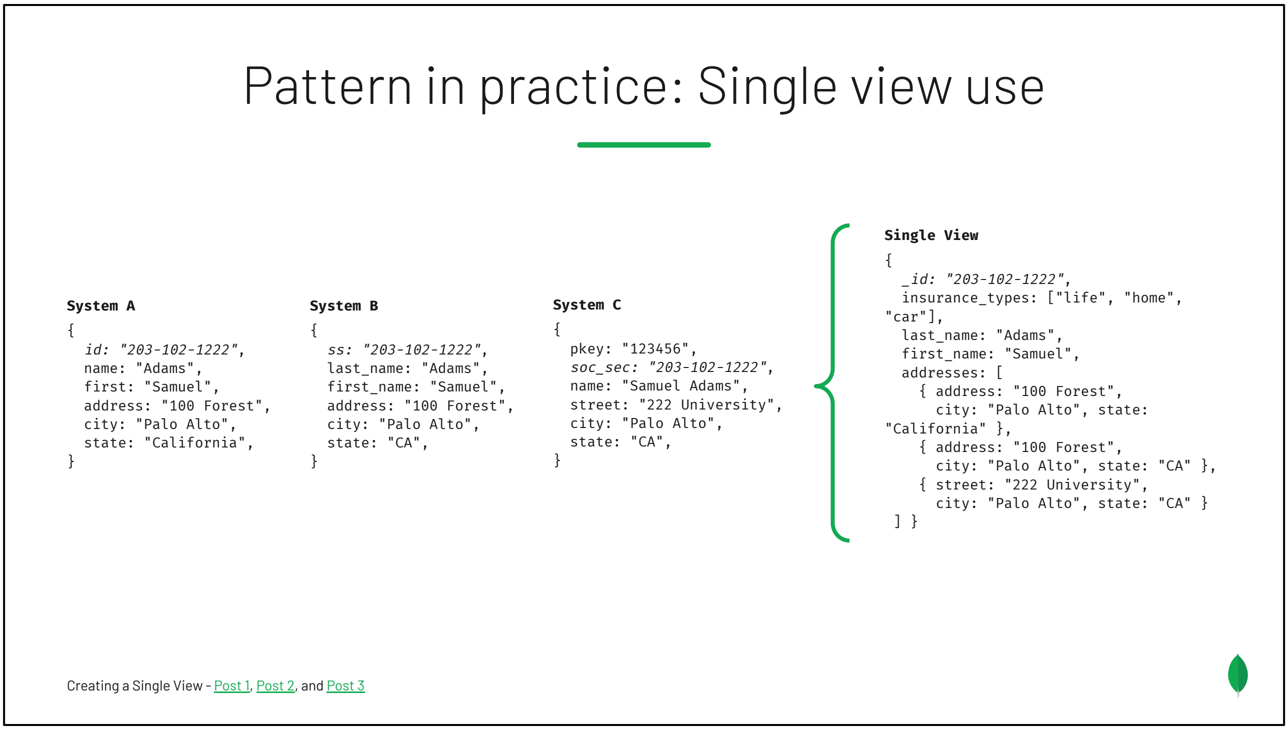
```
{
  "vehicle_type": "truck",
  "owner": "Daniel",
  "taxes": "800",
  "wheels": 10,
  "axles": 3,
}
```

The Polymorphic pattern is useful when there are a variety of documents that have more similarities than differences and the documents need to be kept in a single collection.

An example of this would be if we had a vehicles collection with various different types of vehicle such as cars, trucks, motorbikes or even construction vehicles.

On the slide here we can see that whilst there are some common similarities between trucks and cars that there are some differences such as the number of axles but that overall the differences are minor.

When I send the slide deck out after class, you'll see a link to Patterns blog post on the Polymorphic Pattern https://developer.mongodb.com/how-to/polymorphic-pattern. Feel free to visit that link to learn more about the polymorphic pattern.



Pattern in practice: Single view use

System A

```
{
  id: "203-102-1222",
  name: "Adams",
  first: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "California",
}
```

System B

```
{
  ss: "203-102-1222",
  last_name: "Adams",
  first_name: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "CA",
}
```

System C

```
{
  pkey: "123456",
  soc_sec: "203-102-1222",
  name: "Samuel Adams",
  street: "222 University",
  city: "Palo Alto",
  state: "CA",
}
```

Single View

```
{
  _id: "203-102-1222",
  insurance_types: ["life", "home",
"car"],
  last_name: "Adams",
  first_name: "Samuel",
  addresses: [
    { address: "100 Forest",
      city: "Palo Alto", state:
"California" },
    { address: "100 Forest",
      city: "Palo Alto", state: "CA" },
    { street: "222 University",
      city: "Palo Alto", state: "CA" }
 ] }
```

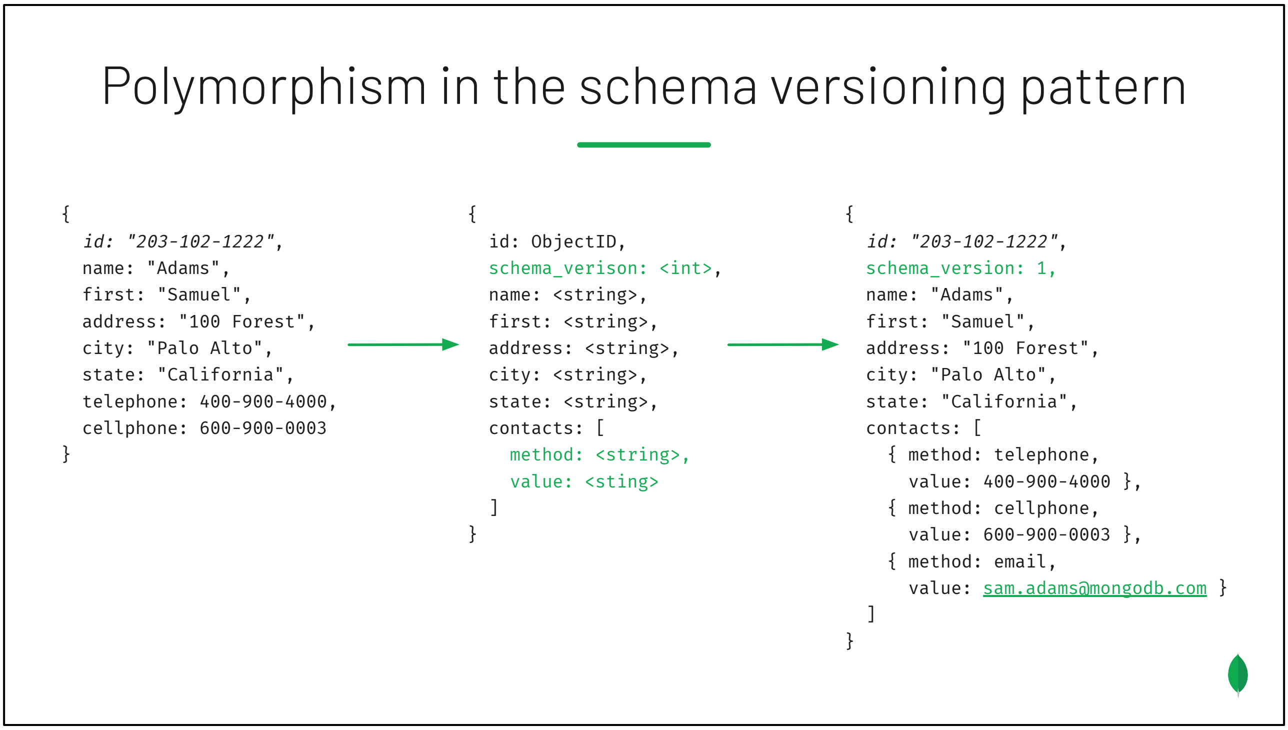
Creating a Single View - Post 1, Post 2, and Post 3

The polymorphic pattern can also be used where you might want to consolidate multiple different collections with the same or similar information into a single collection.

A typical example as shown on the slide is where records for a single customer are stored in various different databases or collections, by using the single view pattern it is possible to collect all of the information into a single representation and document. In the example above, the documents represented life, home, and car insurance customer records which can now be stored in a single consolidated customer insurance document.

This “Single View Use” has been a very successful use case for MongoDB as it allows many different data sources to be unified into a single collection.
https://www.mongodb.com/blog/post/10-step-methodology-single-view-part-1
https://www.mongodb.com/blog/post/10-step-methodology-single-view-part-2
https://www.mongodb.com/blog/post/10-step-methodology-to-creating-a-single-view-of-your-business-part-3

This pattern and indeed the methodology in the blog posts listed on the slide can resolve the generic problem of managing disconnected and duplicate data. The end collection is also a great starting point for further analytics across the now consolidated data.



Polymorphism in the schema versioning pattern

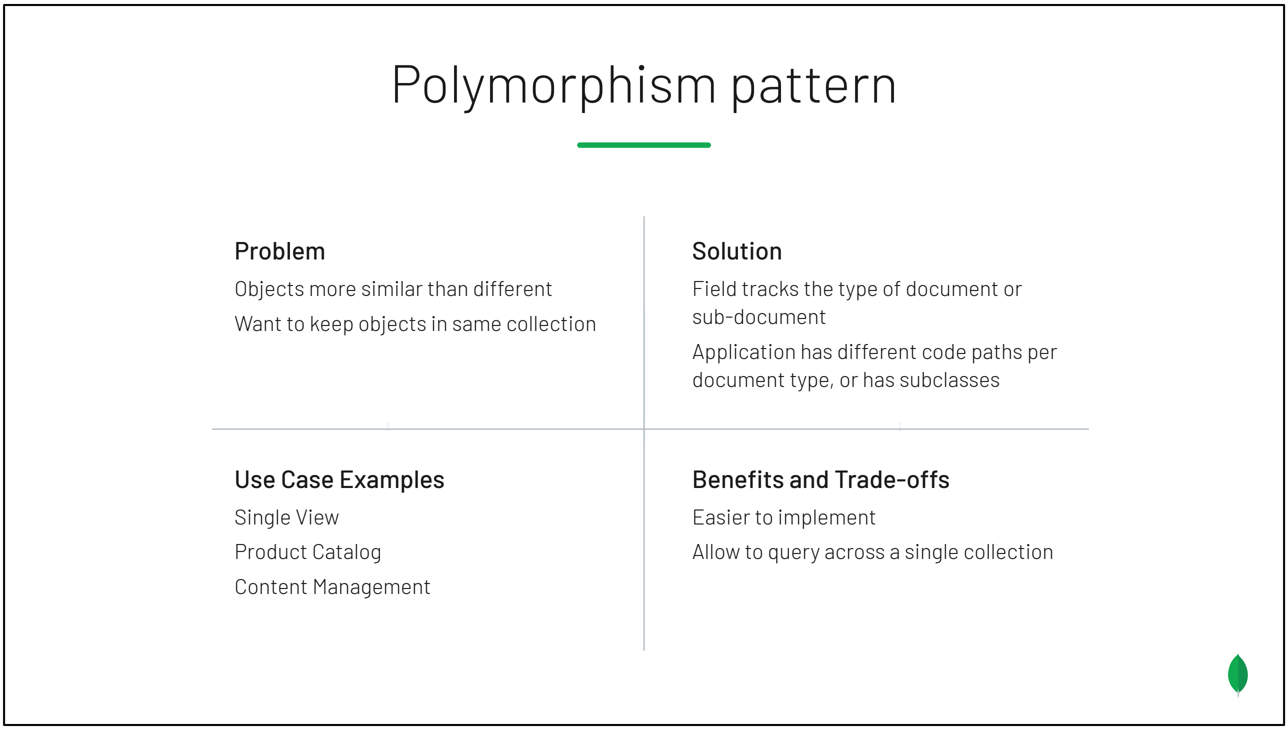
```
{
  id: "203-102-1222",
  name: "Adams",
  first: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "California",
  telephone: 400-900-4000,
  cellphone: 600-900-0003
}
```

```
{
  id: ObjectID,
  schema_verison: <int>,
  name: <string>,
  first: <string>,
  address: <string>,
  city: <string>,
  state: <string>,
  contacts: [
    method: <string>,
    value: <sting>
  ]
}
```

```
{
  id: "203-102-1222",
  schema_version: 1,
  name: "Adams",
  first: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "California",
  contacts: [
    { method: telephone,
      value: 400-900-4000 },
    { method: cellphone,
      value: 600-900-0003 },
    { method: email,
      value: sam.adams@mongodb.com }
  ]
}
```

We'll cover the schema versioning pattern next but in advance, we'll highlight how the polymorphism pattern is a key element to that pattern. The contacts sub-document is an example of applying the polymorphism pattern with a second pattern.

When I send the slide deck out after class, you'll see a link to Patterns blog post on the Schema Versioning Pattern https://www.mongodb.com/blog/post/building-with-patterns-the-schema-versioning-pattern. Feel free to visit that link to learn more about the schema versioning pattern.



Polymorphism pattern

Problem

Objects more similar than different

Want to keep objects in same collection

Solution

Field tracks the type of document or sub-document

Application has different code paths per document type, or has subclasses

Use Case Examples

Single View

Product Catalog

Content Management

Benefits and Trade-offs

Easier to implement

Allow to query across a single collection

We’ll summarise each of patterns under the same four headings of Problem, Solution, Use Case Examples, and Benefits & Trade-offs.

In the case of the Polymorphism Pattern, the problem is that the objects are often more similar than different and we still wish to keep them in the same collection.

The solution is to create a field that track the sub-document, it does however require application paths.

In terms of use case examples, the single view we covered but equally this pattern could be applied to product catalogues or to content management use cases.

In terms of the benefits and trade-offs, the polymorphism is a easy to implement and allows the query to work on a single collection.

When I send the slide deck out after class, you’ll see a link to Patterns blog post on the Polymorphic Pattern https://developer.mongodb.com/how-to/polymorphic-pattern. Feel free to visit that link to learn more about the polymorphic pattern.

Quiz

Quiz

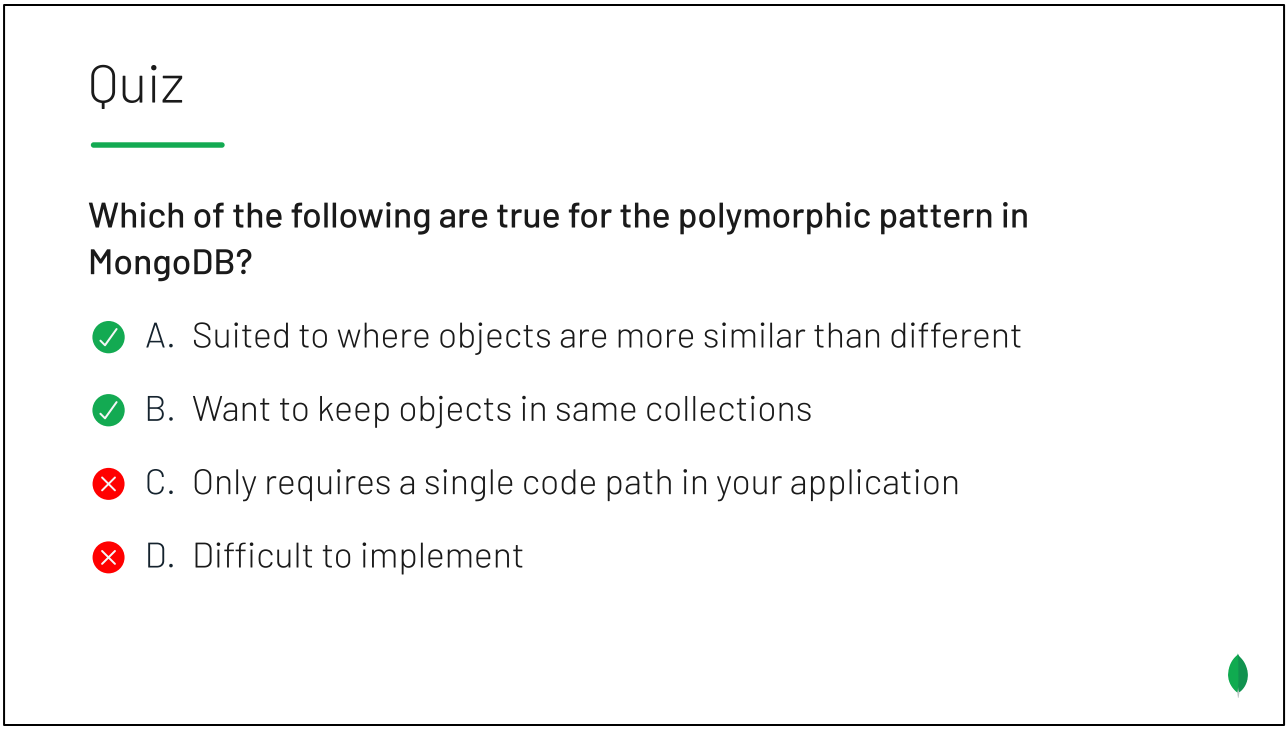
Which of the following are true for the polymorphic pattern in MongoDB? More than 1 answer choice can be correct.

● A. Suited to where objects are more similar than different

● B. Want to keep objects in same collections

● C. Only requires a single code path in your application

● D. Difficult to implement



Quiz

Which of the following are true for the polymorphic pattern in MongoDB?

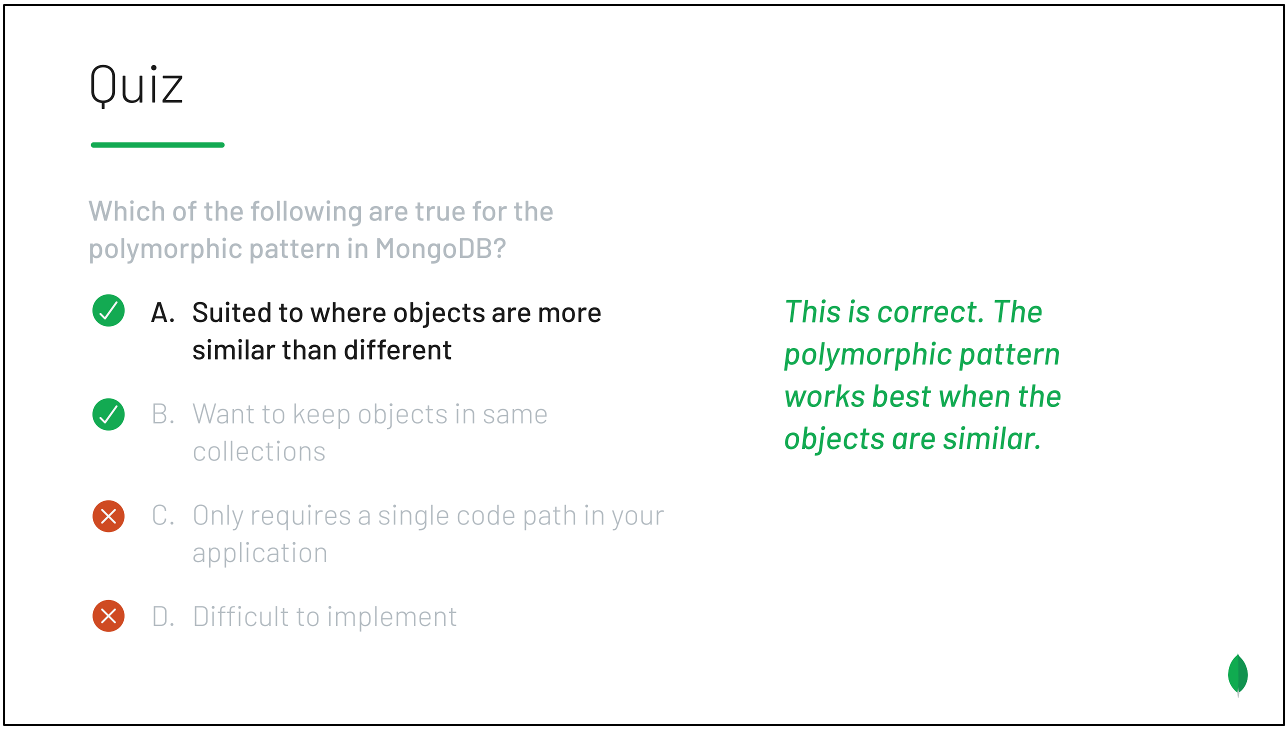
✓ A. Suited to where objects are more similar than different

✓ B. Want to keep objects in same collections

✗ C. Only requires a single code path in your application

✗ D. Difficult to implement

CORRECT: Suited to where objects are more similar than different
CORRECT: Want to keep objects in same collections
INCORRECT: Only requires a single code path in your application - This is incorrect, you will typically need to add a number of code paths to deal with the various differences.
INCORRECT: Difficult to implement - This is incorrect, this pattern is very easy to implement.



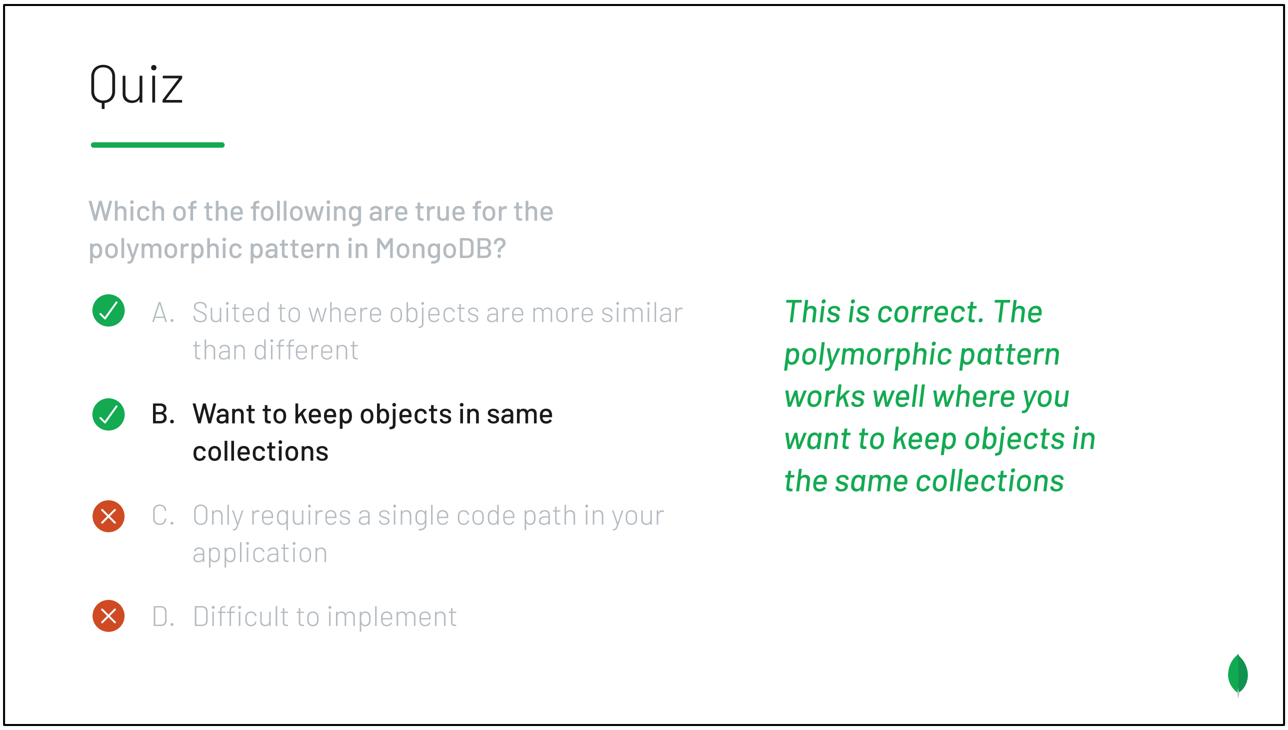
Quiz

Which of the following are true for the polymorphic pattern in MongoDB?

- ✓ A. Suited to where objects are more similar than different
- ✓ B. Want to keep objects in same collections
- ✗ C. Only requires a single code path in your application
- ✗ D. Difficult to implement

This is correct. The polymorphic pattern works best when the objects are similar.

CORRECT: Suited to where objects are more similar than different - This is correct. The polymorphic pattern works best when the objects are similar.



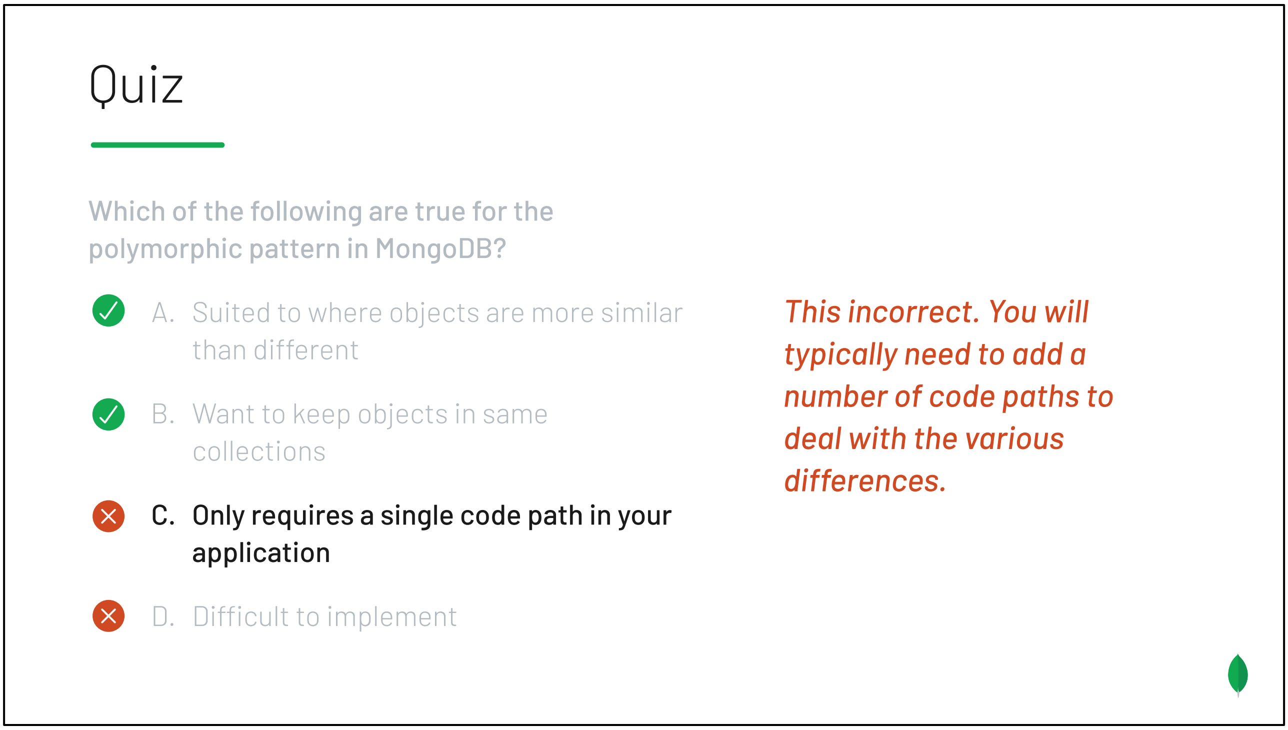
Quiz

Which of the following are true for the polymorphic pattern in MongoDB?

- ✓ A. Suited to where objects are more similar than different
- ✓ B. Want to keep objects in same collections
- ✗ C. Only requires a single code path in your application
- ✗ D. Difficult to implement

This is correct. The polymorphic pattern works well where you want to keep objects in the same collections

CORRECT: Want to keep objects in same collections. - This is correct. The polymorphic pattern works well where you want to keep objects in the same collections.



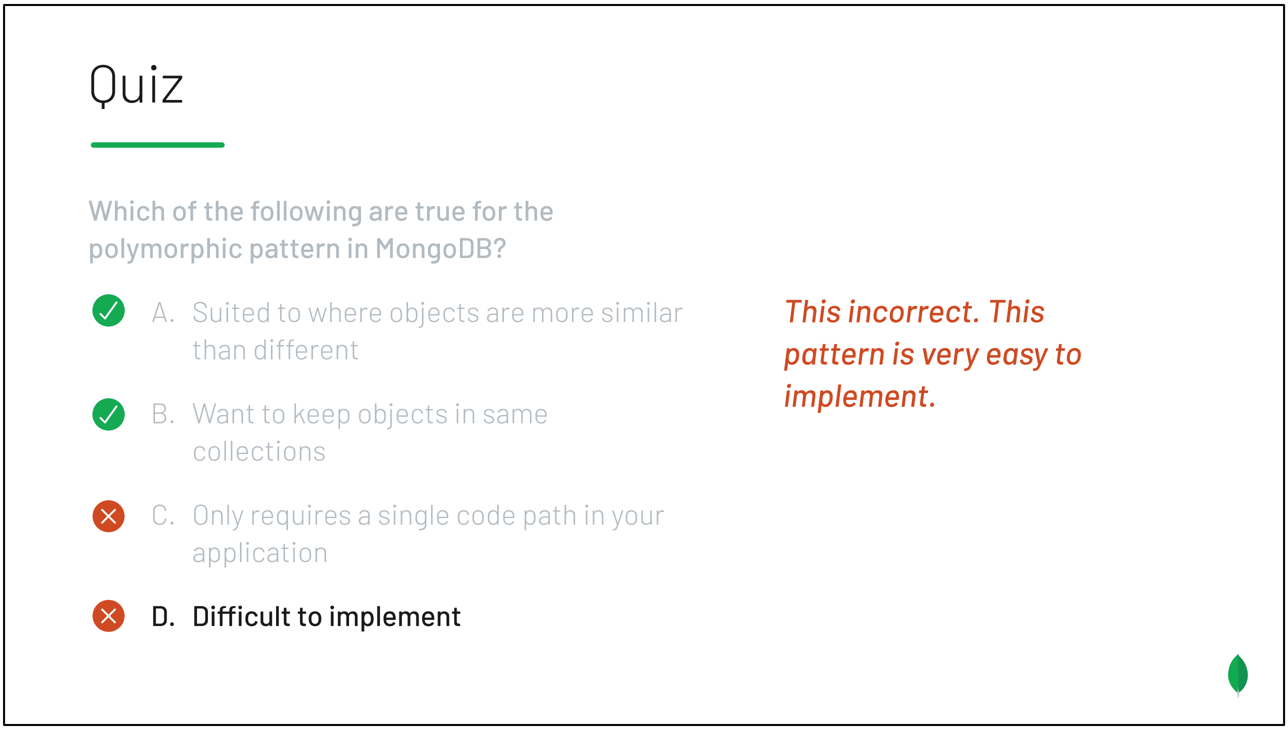
Quiz

Which of the following are true for the polymorphic pattern in MongoDB?

- ✓ A. Suited to where objects are more similar than different
- ✓ B. Want to keep objects in same collections
- ✗ C. Only requires a single code path in your application
- ✗ D. Difficult to implement

This incorrect. You will typically need to add a number of code paths to deal with the various differences.

INCORRECT: Only requires a single code path in your application - This is incorrect. You will typically need to add a number of code paths to deal with the various differences.



Quiz

Which of the following are true for the polymorphic pattern in MongoDB?

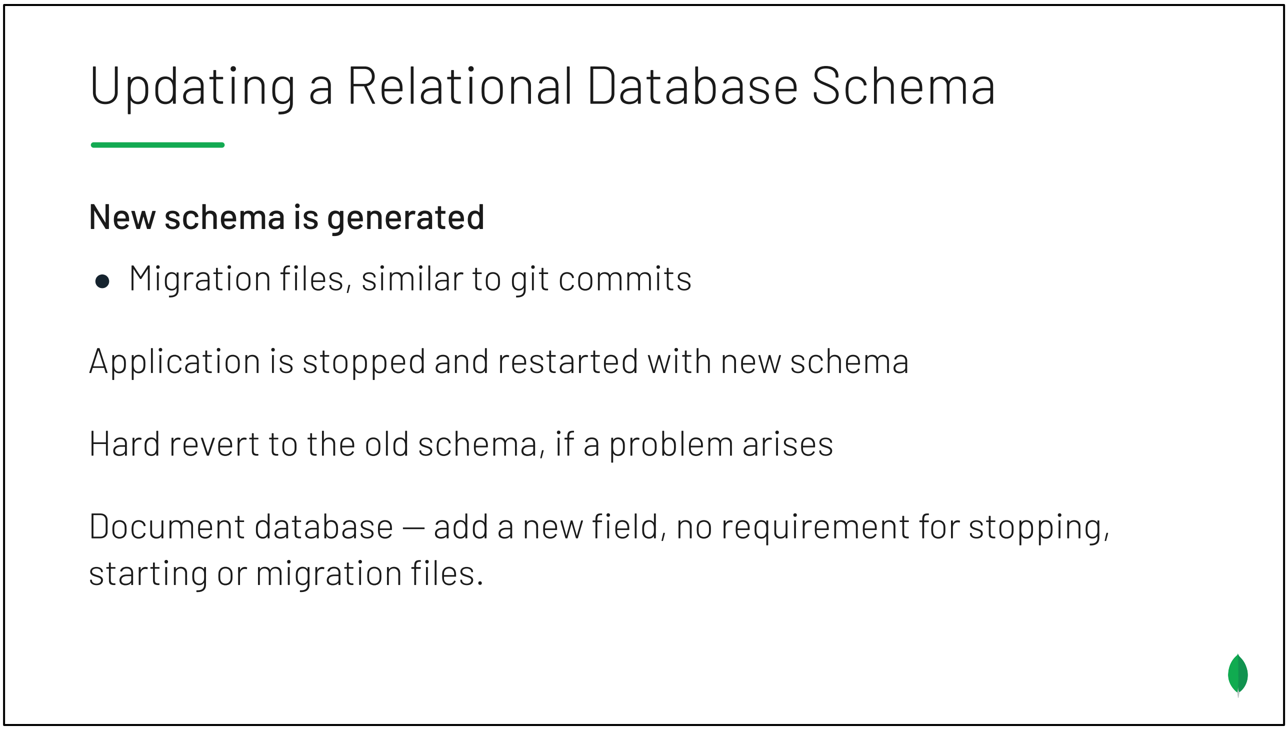
- ✓ A. Suited to where objects are more similar than different
- ✓ B. Want to keep objects in same collections
- ✗ C. Only requires a single code path in your application
- ✗ **D. Difficult to implement**

This incorrect. This pattern is very easy to implement.

INCORRECT: Difficult to implement - This is incorrect. This pattern is very easy to implement.

Schema Versioning Pattern

Let’s look next at the Schema Versioning Pattern, this is a really useful pattern as it utilises the dynamic nature of schemas in documents to support a common task in application lifecycle, how best to manage your database schema when you update your application.



Updating a Relational Database Schema

New schema is generated

- Migration files, similar to git commits

Application is stopped and restarted with new schema

Hard revert to the old schema, if a problem arises

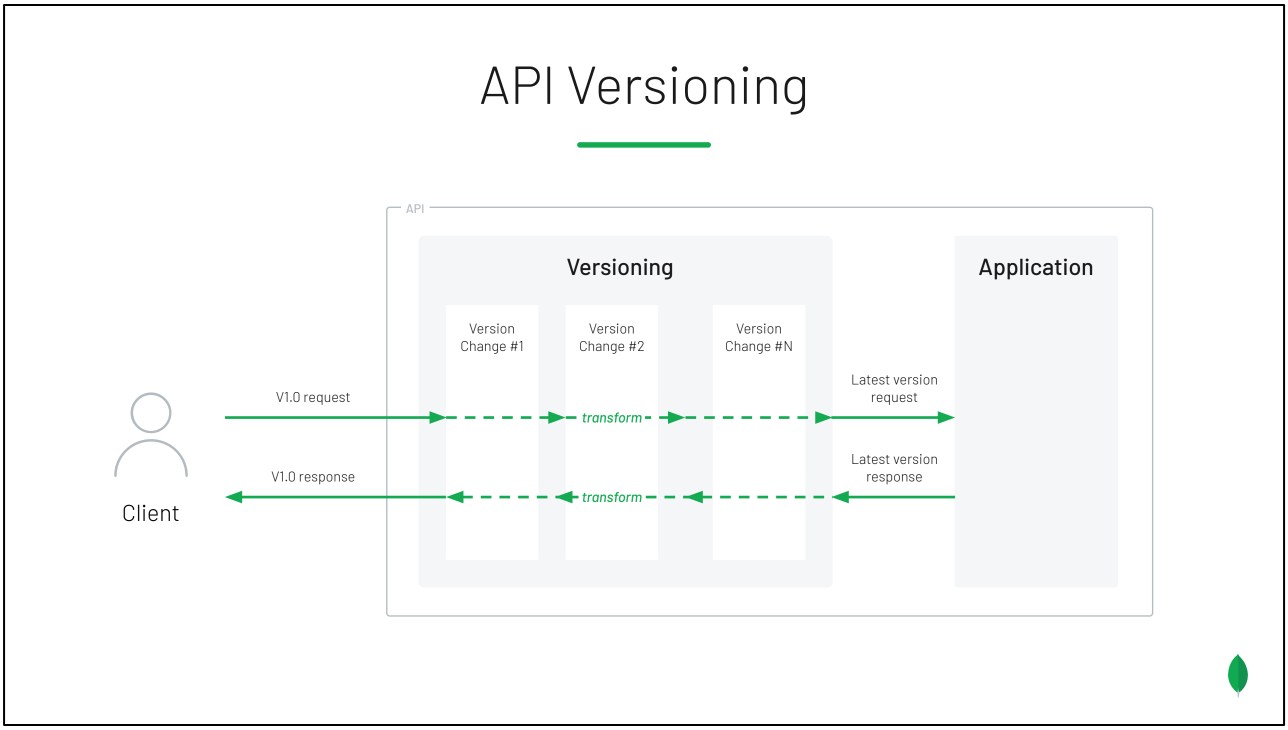
Document database — add a new field, no requirement for stopping, starting or migration files.

A long established issue in relational databases is how do you upgrade the database schema. Typically, you generate the new schema and then use migration files. Your application must be stopped and restarted with the new schema.

It is somewhat difficult to revert back to the previous schema, if a problem arises.

In the case of a document database such as MongoDB, this is not an issue as you can simply add a new field without requiring migration files or restarting the database.

The schema versioning pattern was created to easily allow multiple versions of the schema to be maintained in the database. We'll look at how API versioning is handled as there are similarities between this and the pattern.



API Versioning

API

Versioning

Application

Version Change #1

Version Change #2

Version Change #N

Client

V1.0 request

V1.0 response

transform

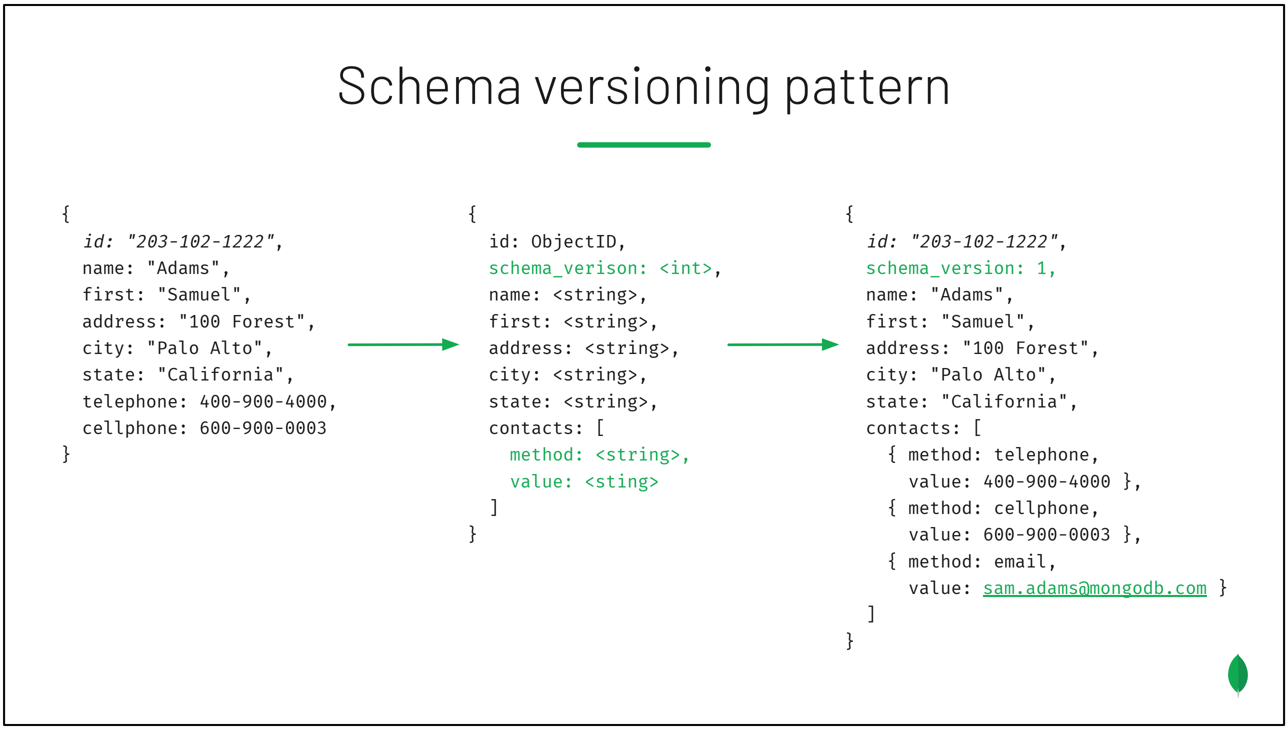
transform

Latest version request

Latest version response

There are various approaches to dealing with different versions of an API. A good example of how this can be handled is available in this blog by Intercom. Essentially, a request from a client using an old API version is sent to the application server. The request is passed up through various transformation layers until it reaches the latest version which is then process and the result is likewise passed back down and transformed through accordingly.

When I send the slide deck out after class, you'll see a link to Intercom Engineering blog post on API versioning Inspired by https://www.intercom.com/blog/api-versioning/. Feel free to visit that link to learn more about API versioning and how Intercom designed their approach.



Schema versioning pattern

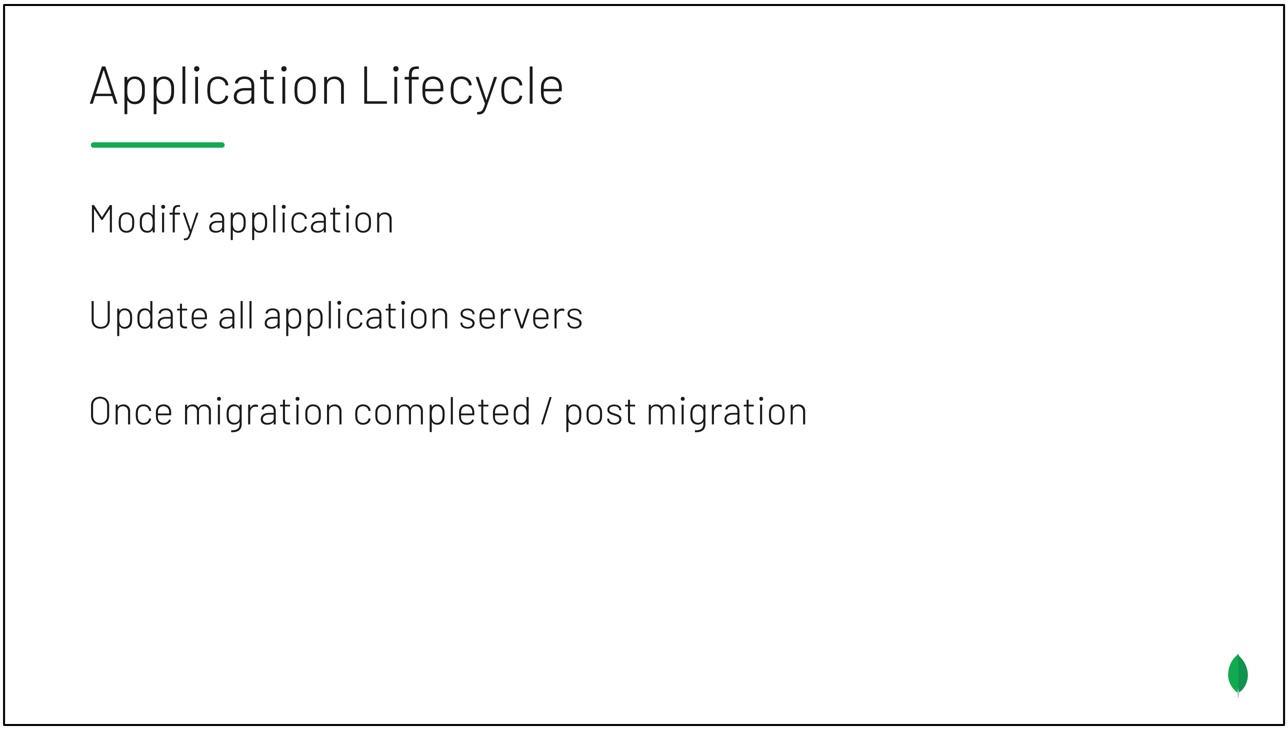
```
{
  id: "203-102-1222",
  name: "Adams",
  first: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "California",
  telephone: 400-900-4000,
  cellphone: 600-900-0003
}
```

```
{
  id: ObjectID,
  schema_verison: <int>,
  name: <string>,
  first: <string>,
  address: <string>,
  city: <string>,
  state: <string>,
  contacts: [
    method: <string>,
    value: <sting>
  ]
}
```

```
{
  id: "203-102-1222",
  schema_version: 1,
  name: "Adams",
  first: "Samuel",
  address: "100 Forest",
  city: "Palo Alto",
  state: "California",
  contacts: [
    { method: telephone,
      value: 400-900-4000 },
    { method: cellphone,
      value: 600-900-0003 },
    { method: email,
      value: sam.adams@mongodb.com }
  ]
}
```

The schema versioning pattern adds a new schema_version field, if the version changes then the value for the version is incremented. This allows for the application to adjust to changes in the schema easily. This approach allows for schema upgrades without downtime. It also allows applications to support multiple versions (say the latest and older versions) within the application's code.

When I send the slide deck out after class, you'll see a link to Patterns blog post on the Schema Versioning Pattern https://www.mongodb.com/blog/post/building-with-patterns-the-schema-versioning-pattern. Feel free to visit that link to learn more about the schema versioning pattern.



Application Lifecycle

Modify application

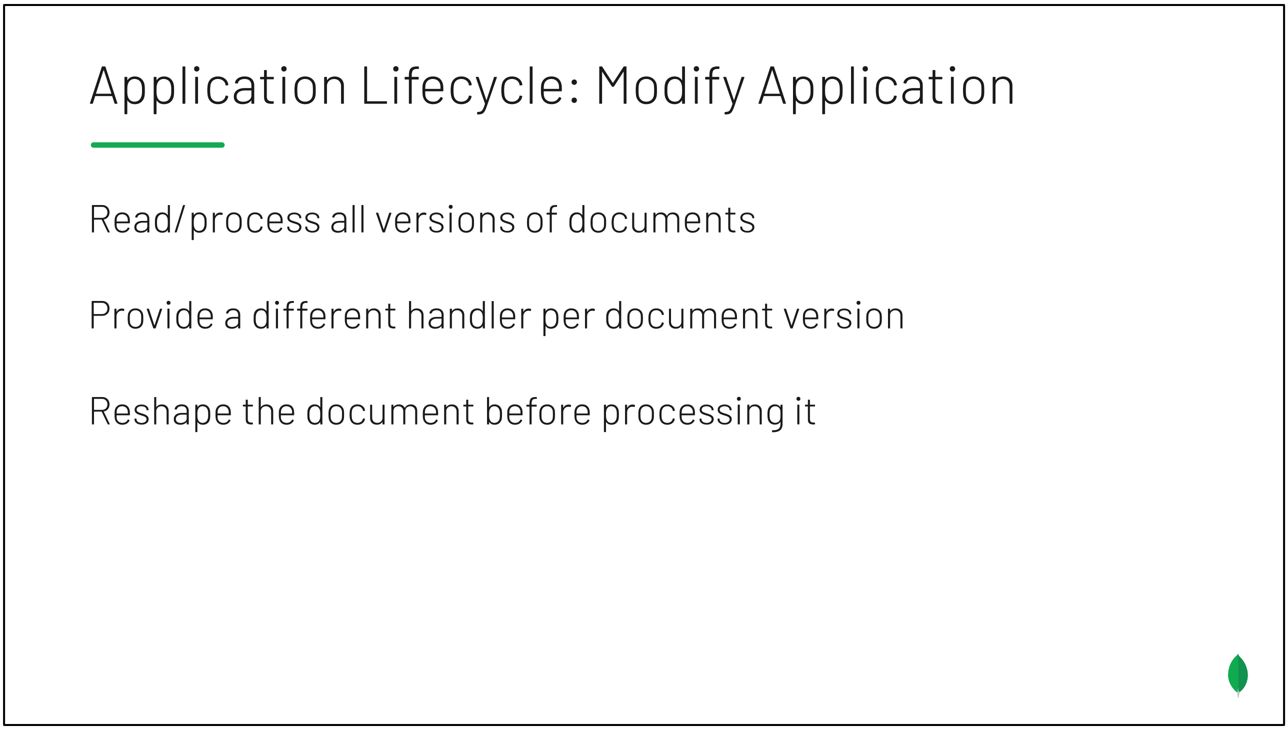
Update all application servers

Once migration completed / post migration

In terms of a typical application lifecycle, there are a number of major stages. Typically, you modify the application,

then you update the servers where it is running and

then you cleanup the code that you left to maintain compatibility with the older versions whilst you were migrating.



Application Lifecycle: Modify Application

Read/process all versions of documents

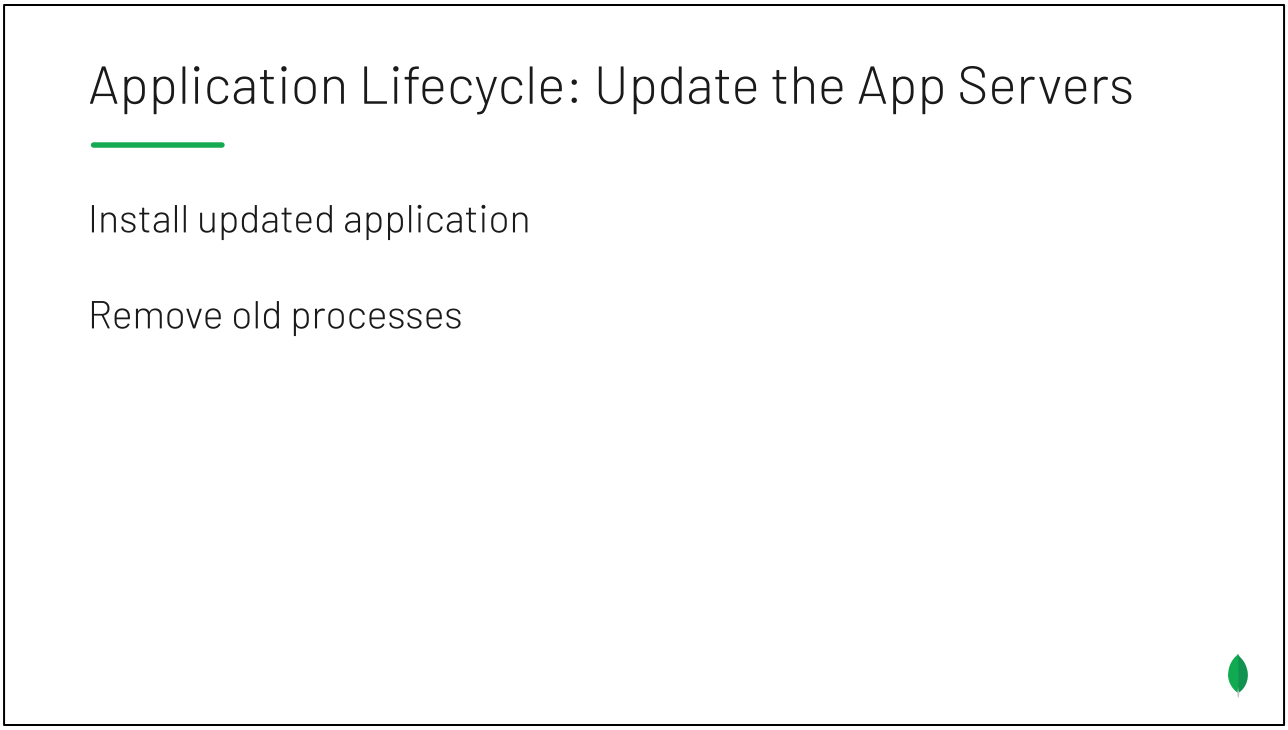
Provide a different handler per document version

Reshape the document before processing it

Let’s look at the first stage, modifying the application, here you would typically read all the versions of the documents,

then you will need to have a different handler per version of the document and

finally you pass the document through all the relevant version handlers which would reshape it before the final processing of it.

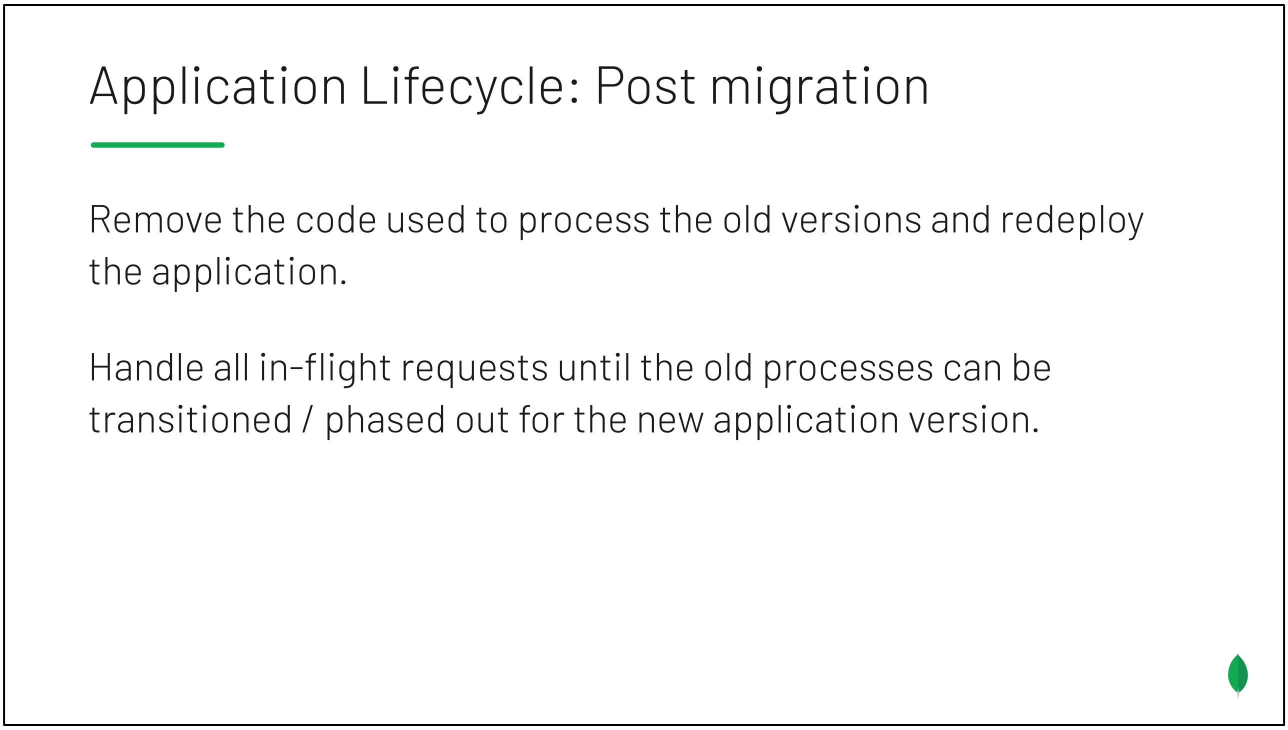


Application Lifecycle: Update the App Servers

Install updated application

Remove old processes

The actual updating stage in the life cycle involves installing the updated application to the various application servers and swapping over to new versions of these by running processes and gracefully removing any older processes, stopping new requests to them and redirecting them to the new processes. Once all the in-flight requests are handled you can then fully remove the old processes as the new processes will seamlessly take over at that point.

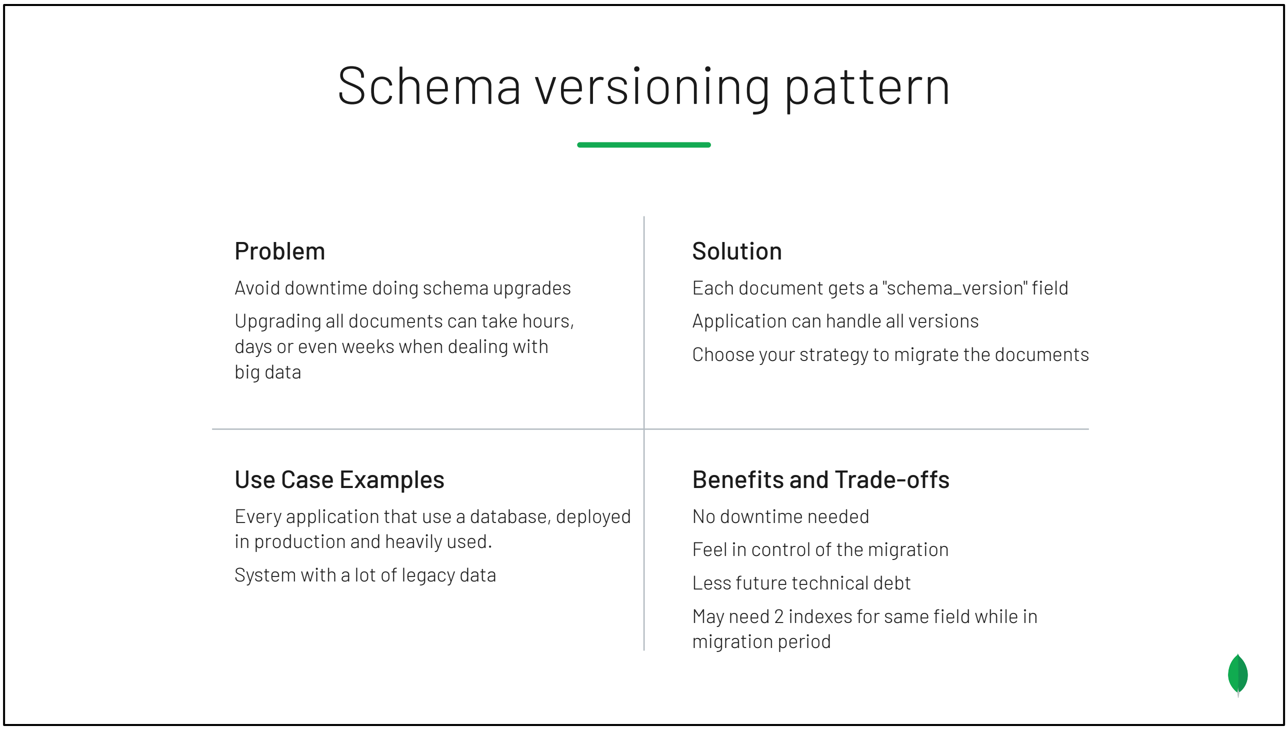


Application Lifecycle: Post migration

Remove the code used to process the old versions and redeploy the application.

Handle all in-flight requests until the old processes can be transitioned / phased out for the new application version.

The actual updating stage in the life cycle involves installing the updated application to the various application servers and swapping over to new versions of these by running processes and gracefully removing any older processes, stopping new requests to them and redirecting them to the new processes. Once all the in-flight requests are handled you can then fully remove the old processes as the new processes will seamlessly take over at that point.



Schema versioning pattern

Problem

Avoid downtime doing schema upgrades

Upgrading all documents can take hours, days or even weeks when dealing with big data

Solution

Each document gets a "schema_version" field

Application can handle all versions

Choose your strategy to migrate the documents

Use Case Examples

Every application that use a database, deployed in production and heavily used.

System with a lot of legacy data

Benefits and Trade-offs

No downtime needed

Feel in control of the migration

Less future technical debt

May need 2 indexes for same field while in migration period

As noted previously we'll summarise each of pattern under the same four headings of Problem, Solution, Use Case Examples, and Benefits & Trade-offs.

In the terms of the Schema Versioning Pattern and the "Problem" heading, we can see this pattern helps avoid downtime by allowing your applications to run during schema upgrades. It can limit the amount of documents that need to be upgraded at any point in time and can space out this upgrading to reduce the processing and impact of the document schema upgrade.

In terms of a Solution, each document is assigned a schema_version field and you add code within your application to handle the various versions. This allows you to specifically choose the strategy you wish to take to migrate your documents.

In terms of Use Case Examples, this pattern applies to any heavily used production database. It is also a strong contender in situations where your system has a lot of legacy data.

The key benefits and trade-offs with this pattern are that no downtime is needed, the developers feel in control of the migration and it supports less future technical debt. The caveat is that it may require 2 indexes for the same fields during the migration period.

When I send the slide deck out after class, you'll see a link to Patterns blog post on the Schema Versioning Pattern https://www.mongodb.com/blog/post/building-with-patterns-the-schema-versioning-pattern. Feel free to visit that link to learn more about the schema versioning pattern.

Quiz

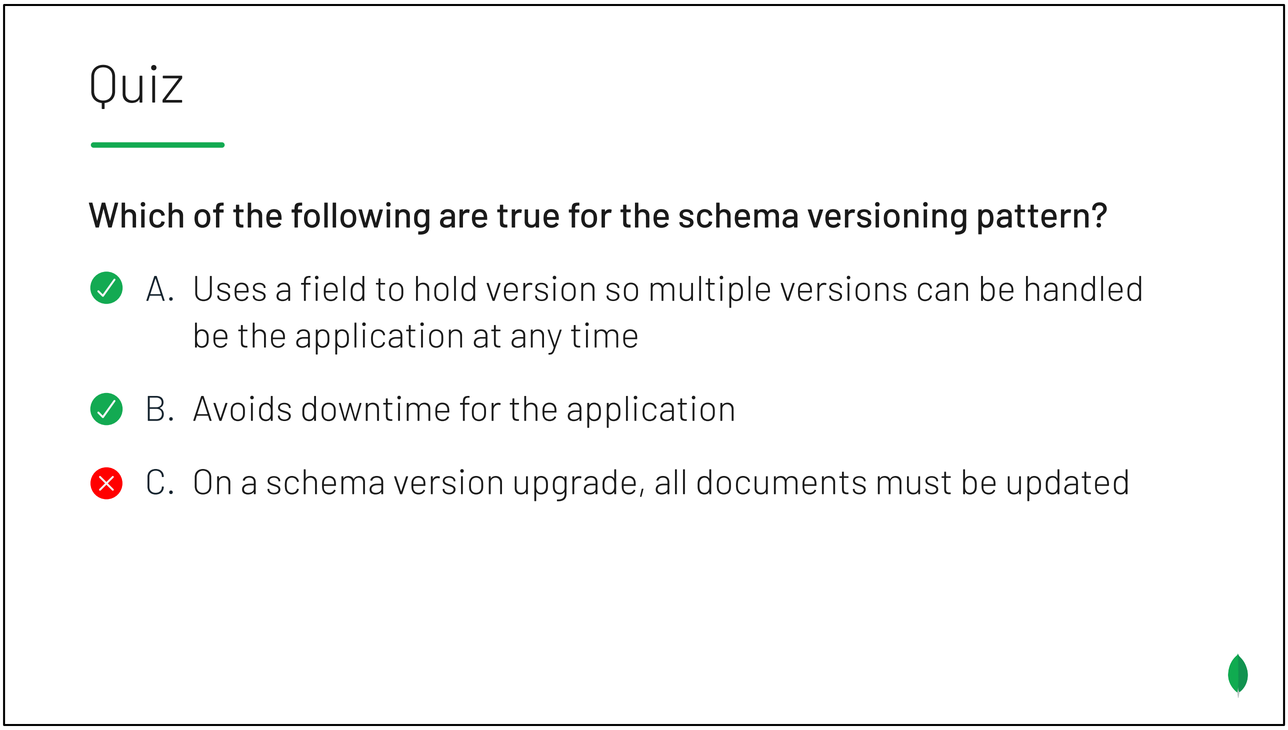
Quiz

Which of the following are true for the schema versioning pattern?
More than 1 answer choice can be correct.

● A. Uses a field to hold version so multiple versions can be handled be the application at any time

● B. Avoids downtime for the application

● C. On a schema version upgrade, all documents must be updated



Quiz

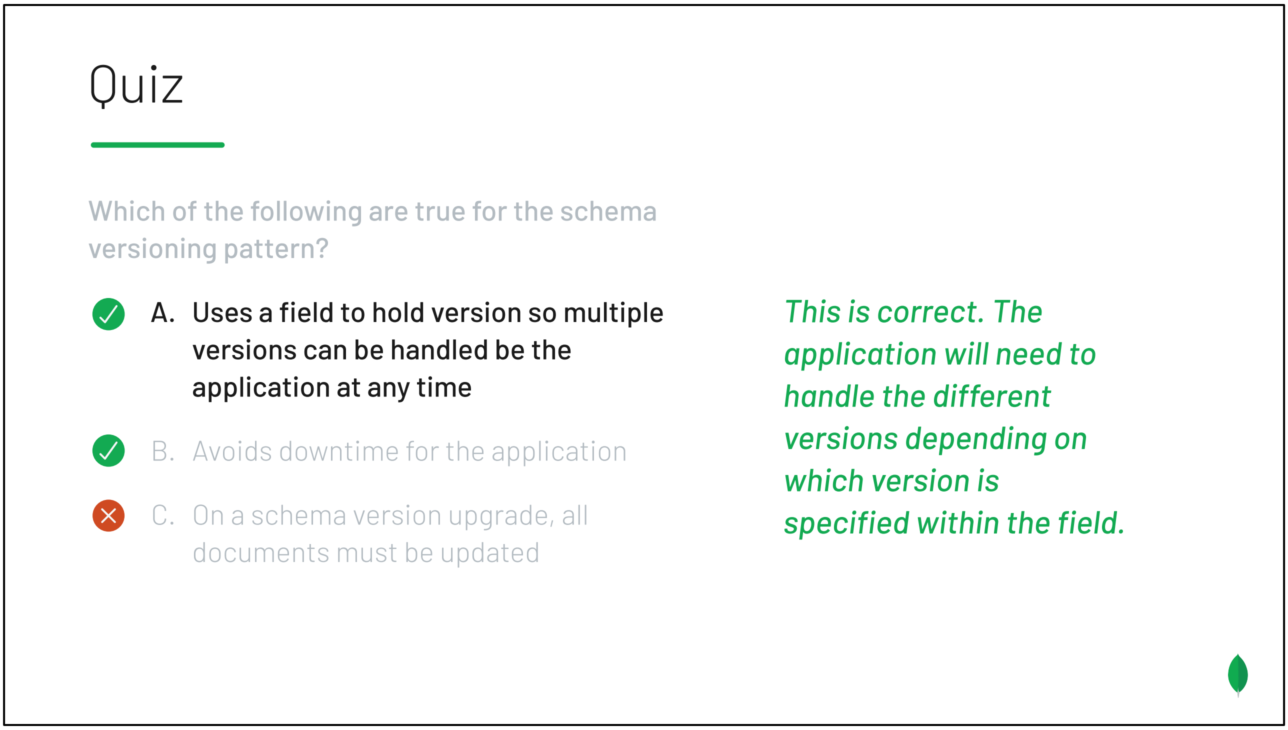
Which of the following are true for the schema versioning pattern?

✓ A. Uses a field to hold version so multiple versions can be handled be the application at any time

✓ B. Avoids downtime for the application

✗ C. On a schema version upgrade, all documents must be updated

CORRECT: Uses a field to hold version so multiple versions can be handled be the application at any time - The application will need to handle the different versions depending on which version is specified within the field
CORRECT: Avoids downtime for the application - Using multiple versions specified by the field allows the application to avoid downtime
INCORRECT: On a schema version upgrade, all documents must be updated - This is incorrect and is true when you don't use this pattern.



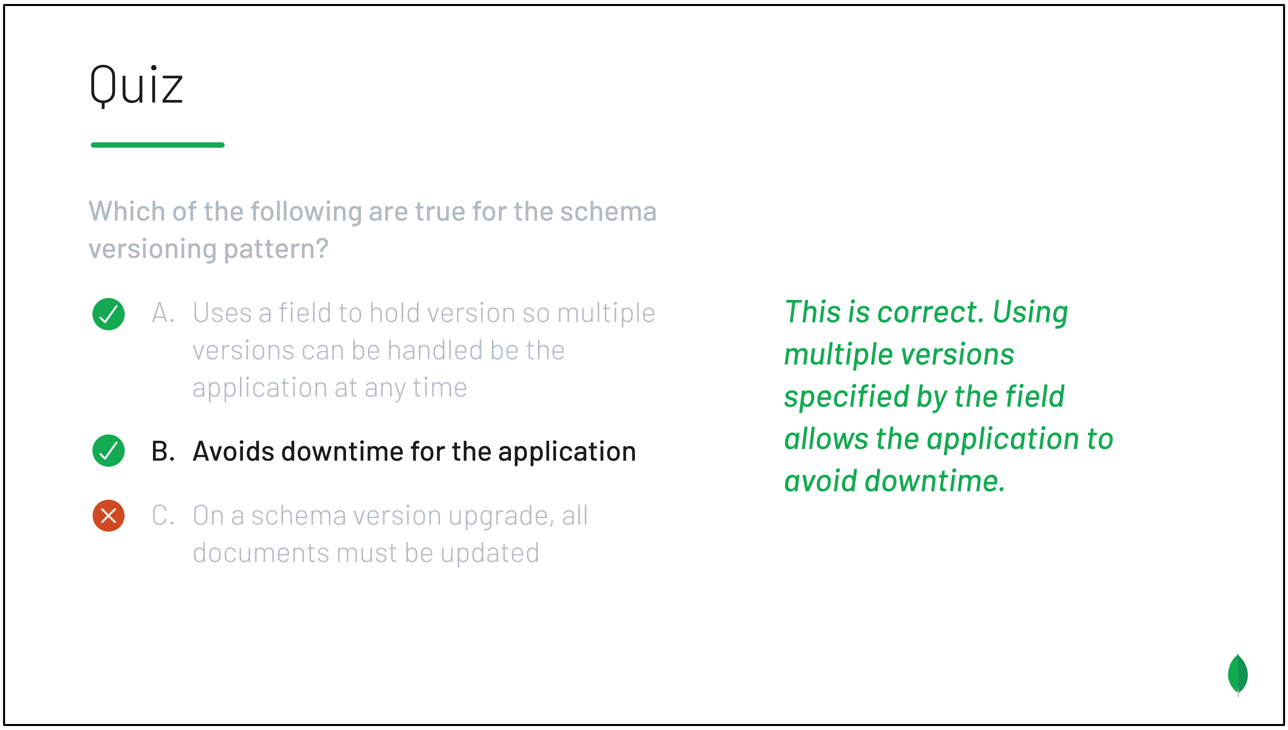
Quiz

Which of the following are true for the schema versioning pattern?

- ✓ A. Uses a field to hold version so multiple versions can be handled be the application at any time
- ✓ B. Avoids downtime for the application
- ✗ C. On a schema version upgrade, all documents must be updated

This is correct. The application will need to handle the different versions depending on which version is specified within the field.

CORRECT: Uses a field to hold version so multiple versions can be handled be the application at any time - This is correct. The application will need to handle the different versions depending on which version is specified within the field



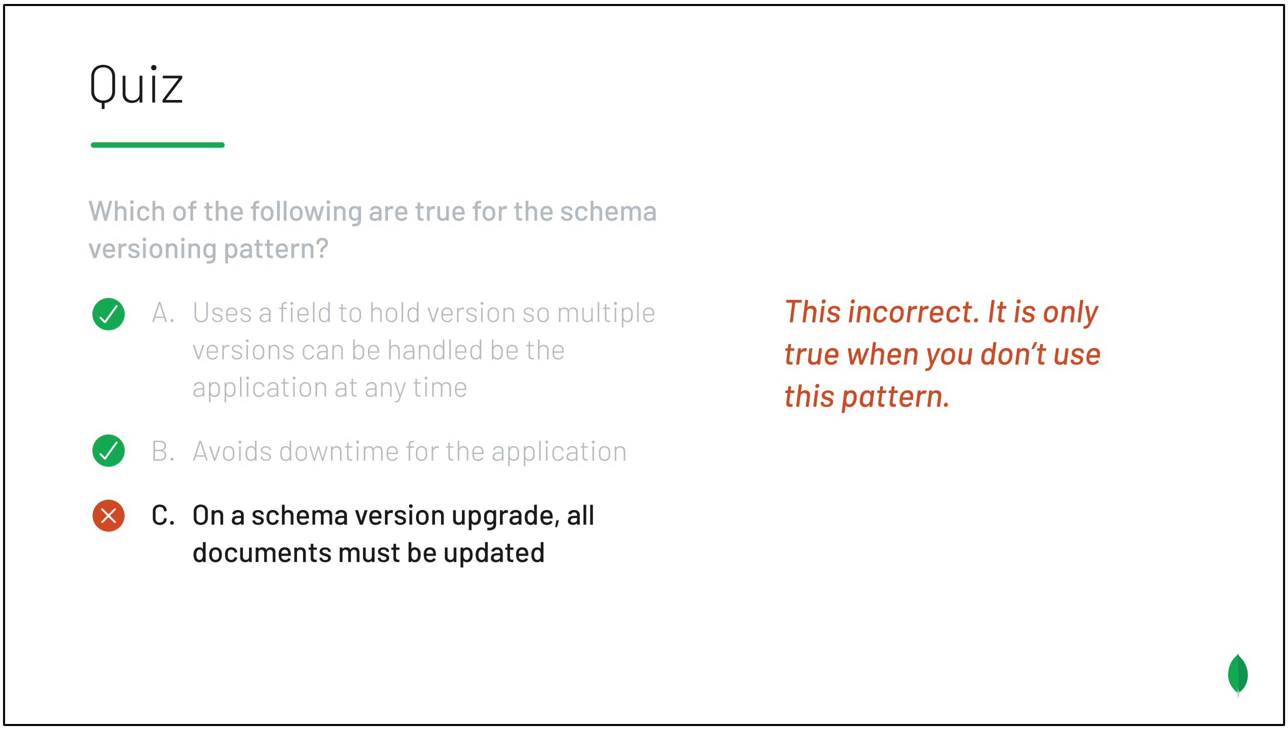
Quiz

Which of the following are true for the schema versioning pattern?

- ✓ A. Uses a field to hold version so multiple versions can be handled be the application at any time
- ✓ B. Avoids downtime for the application
- ✗ C. On a schema version upgrade, all documents must be updated

This is correct. Using multiple versions specified by the field allows the application to avoid downtime.

CORRECT: Avoids downtime for the application - This is correct. Using multiple versions specified by the field allows the application to avoid downtime



Quiz

Which of the following are true for the schema versioning pattern?

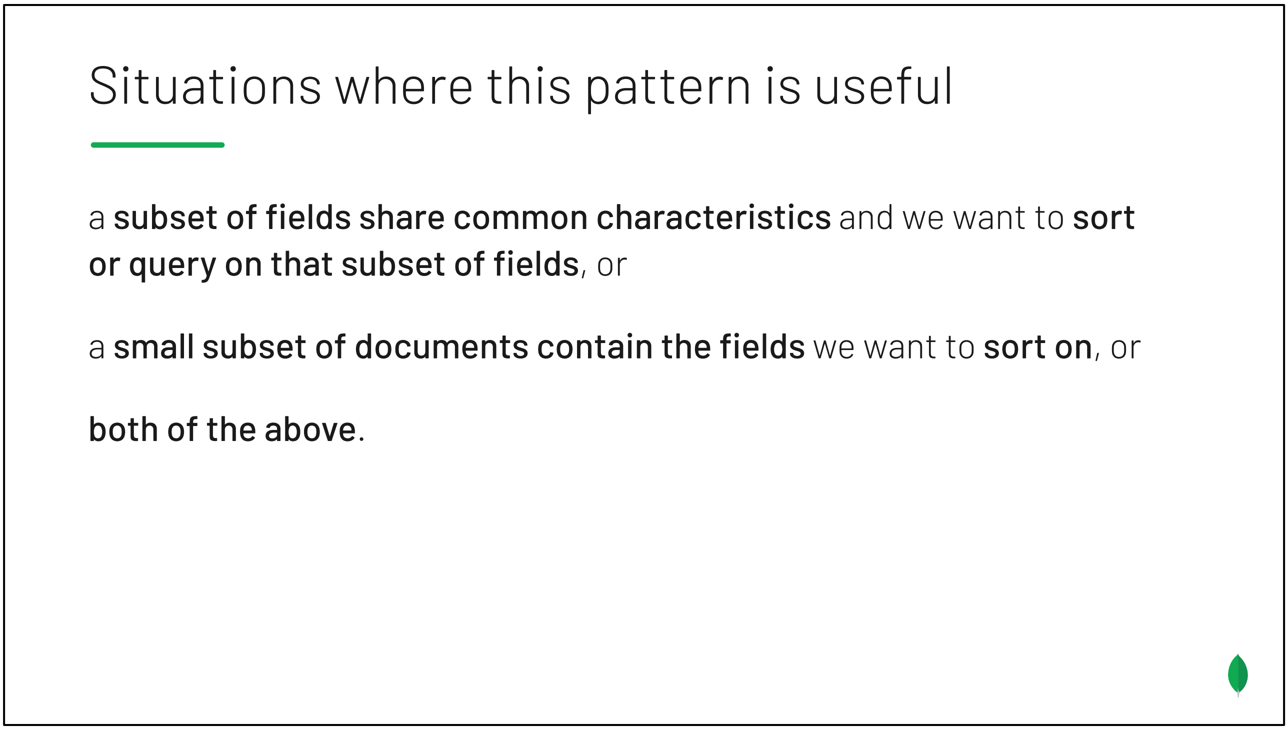
- ✓ A. Uses a field to hold version so multiple versions can be handled be the application at any time
- ✓ B. Avoids downtime for the application
- ✗ C. On a schema version upgrade, all documents must be updated

This incorrect. It is only true when you don’t use this pattern.

INCORRECT: On a schema version upgrade, all documents must be updated - This is incorrect. It is only true when you don’t use this pattern.

Attribute Pattern

Let’s look next at another common pattern, the Attribute Pattern. This is useful where the object being modelling can have similar but not the same fields or information to be modelled.



Situations where this pattern is useful

a **subset of fields share common characteristics** and we want to **sort or query on that subset of fields**, or

a **small subset of documents contain the fields** we want to **sort on**, or

both of the above.

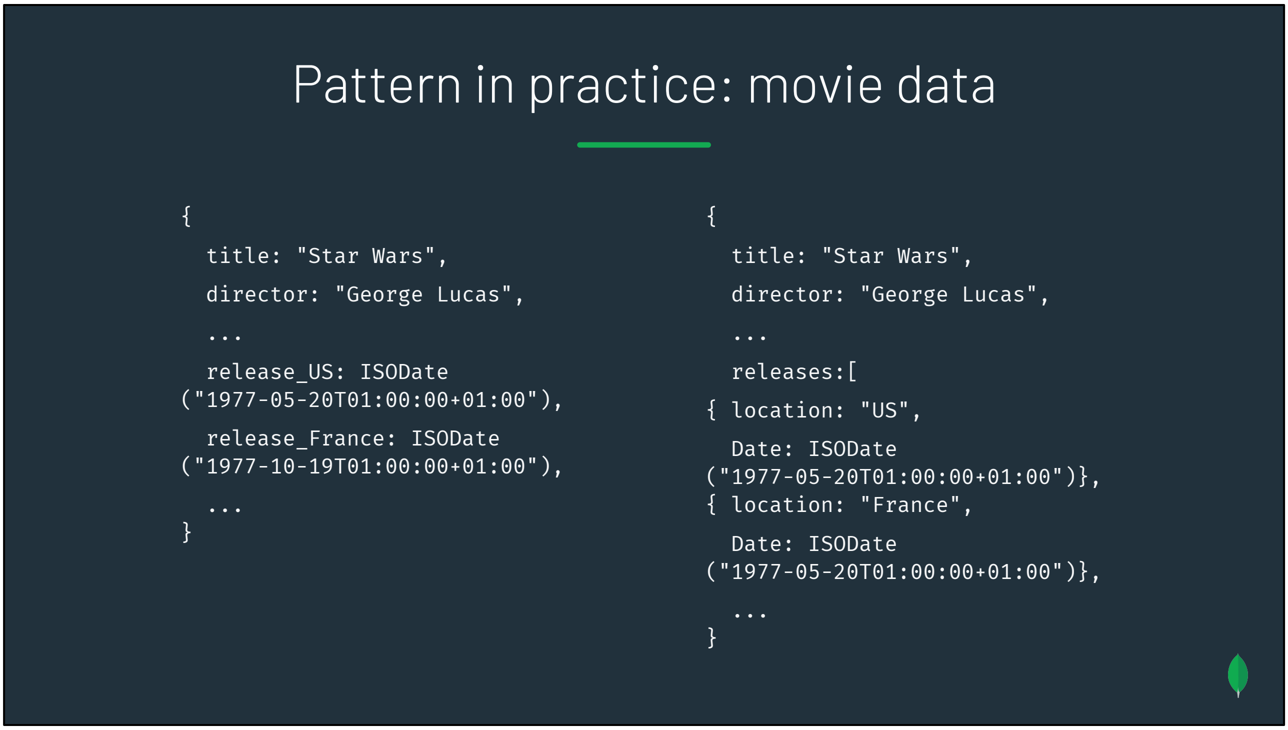
The attribute pattern is a useful in two specific situations, firstly when:

You have a subset of fields that share common characteristics and you want to sort or query on a subset of those fields, OR

You have a small subset of documents that contain the fields you want sort, OR

Both of these situations.

Let’s next look at an example this pattern in practice with movie data.



Pattern in practice: movie data

```
{
  title: "Star Wars",
  director: "George Lucas",
  ...
  release_US: ISODate
("1977-05-20T01:00:00+01:00"),
  release_France: ISODate
("1977-10-19T01:00:00+01:00"),
  ...
}
```

```
{
  title: "Star Wars",
  director: "George Lucas",
  ...
  releases:[
{ location: "US",
  Date: ISODate
("1977-05-20T01:00:00+01:00")},
{ location: "France",
  Date: ISODate
("1977-05-20T01:00:00+01:00")},
  ...
}
```

Taking an example of movie data where we have fields representing the title, the director, the cast, the locations and other details including when it was released.

A typical query would involve sorting results for movies by when they were released. However, a movie isn’t always released simultaneously world wide and that means countries may have differing release dates.

In the snippet of the document we can see a fraction of the releases, just for the US and for France.

In the first iteration of our schema, we use a release_COUNTY field where each country has an individual field representing when the film was released there.

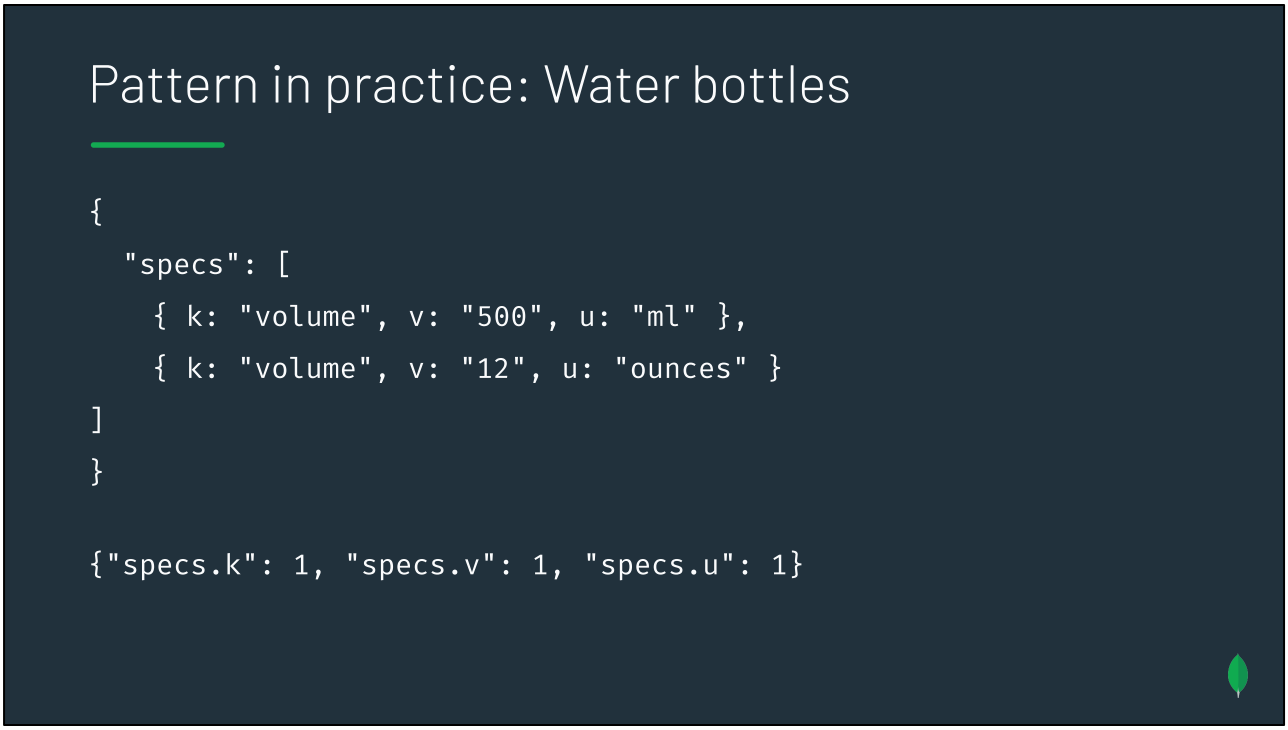
This schema also means we’ll need to add an index per country that we want to query a release on.

Now, let’s apply the Attribute pattern and refactor this schema.

In the new schema using the Attribute pattern, the releases are now stored as documents in the ‘releases’ array with each array item holding a location and a data for the movie’s release there.

A single compound index on releases.location and releases.date is all that is needed with the new schema to service any release related query.

When I send the slide deck out after class, you’ll see a link to Patterns blog post on the Attribute Pattern https://developer.mongodb.com/how-to/attribute-pattern. Feel free to visit that link to learn more about the attribute pattern.



Pattern in practice: Water bottles

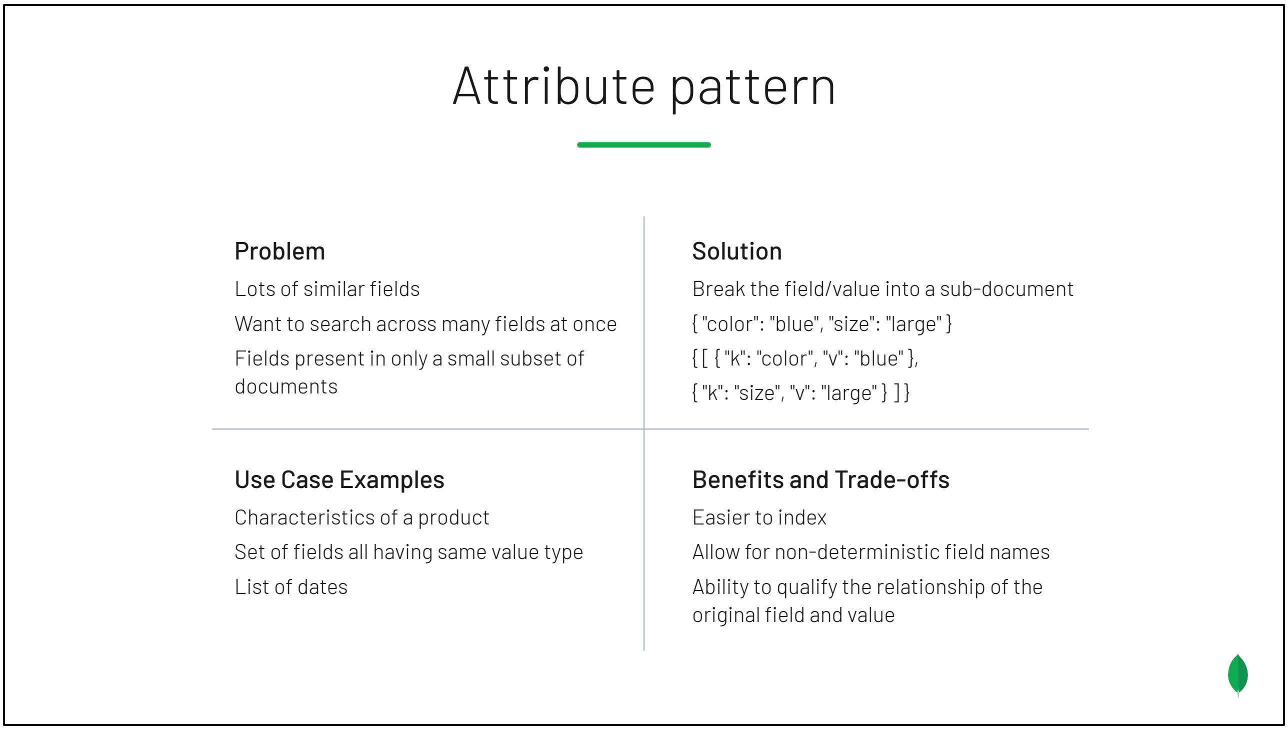
```
{
  "specs": [
    { k: "volume", v: "500", u: "ml" },
    { k: "volume", v: "12", u: "ounces" }
]
}

{"specs.k": 1, "specs.v": 1, "specs.u": 1}
```

Let’s looks at another example, assuming we are storing data around bottles of water in our collection.

We can use the key/value convention to support the use of non-deterministic naming and we can then also easily add qualifiers.

In the example, the key refers to the volume, the value refers to the measure of that volume, and the qualifier is the unit (u) for that volume.

The index can similarly use a compound index on the key, value and unit.



Attribute pattern

Problem

Lots of similar fields

Want to search across many fields at once

Fields present in only a small subset of documents

Solution

Break the field/value into a sub-document

```
{ "color": "blue", "size": "large" }
{ [ { "k": "color", "v": "blue" },
{ "k": "size", "v": "large" } ] }
```

Use Case Examples

Characteristics of a product

Set of fields all having same value type

List of dates

Benefits and Trade-offs

Easier to index

Allow for non-deterministic field names

Ability to qualify the relationship of the original field and value

As noted previously we'll summarise each of pattern under the same four headings of Problem, Solution, Use Case Examples, and Benefits & Trade-offs.

In terms of the problem, there are similar fields in the document and we want to be able to query across many of these fields at once. Additionally, these fields may only be present in a small subset of the documents.

The solution in terms of the attribute to this is to break the similar fields into key values in a sub-document. We can easily add qualifiers to the data in this format. We can use a compound index on these to easily index the data.

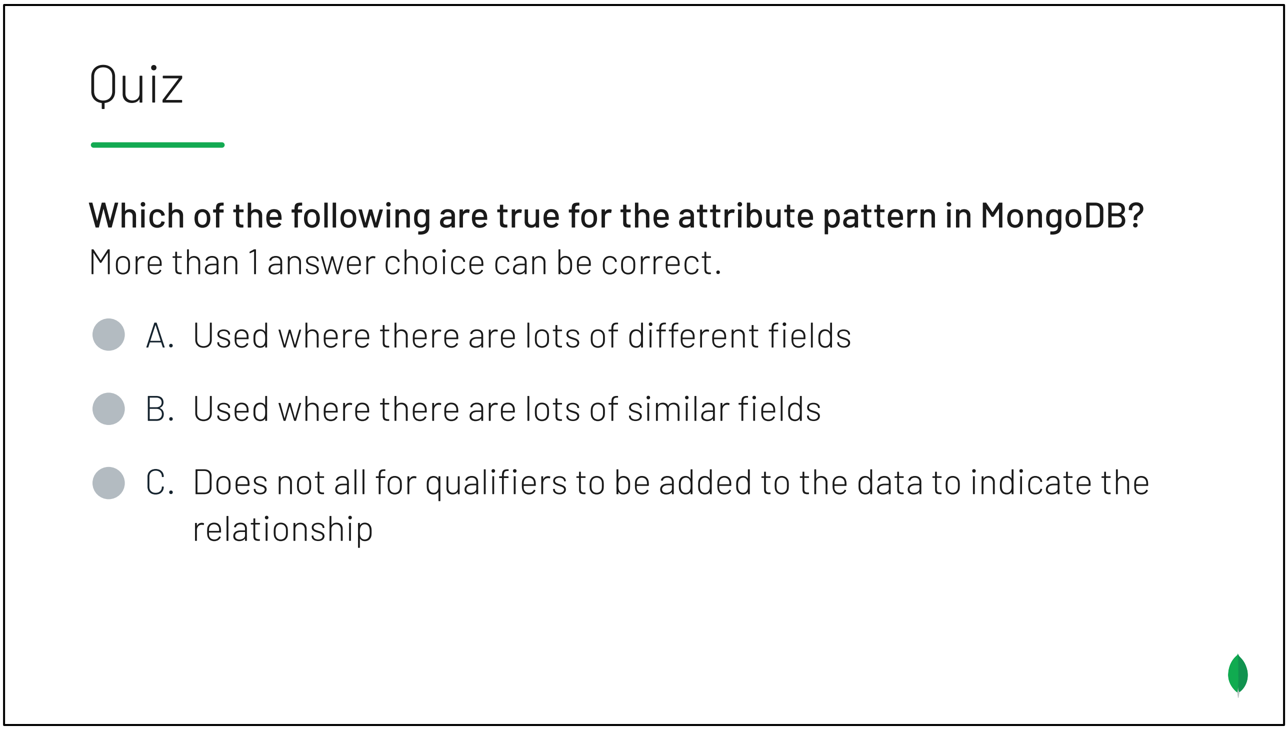
This pattern is typically found where you describe the characteristics of a product (e.g. in e-commerce), or where there are a set of fields all of which have the same value type, or where there is a list of dates.

In terms of benefits, you can use a compound index as noted earlier which makes this easier to index. The pattern also allows for non-deterministic field names and provides the ability to add qualifiers that can further describe the relationship between the original field and value. In our water bottle example, we showed how we could add a unit qualifier to add this additional detail.

When I send the slide deck out after class, you'll see a link to a HowTo blog post on using the Attribute Pattern https://developer.mongodb.com/how-to/attribute-pattern.

Feel free to visit that link to learn more about the attribute pattern.

Quiz



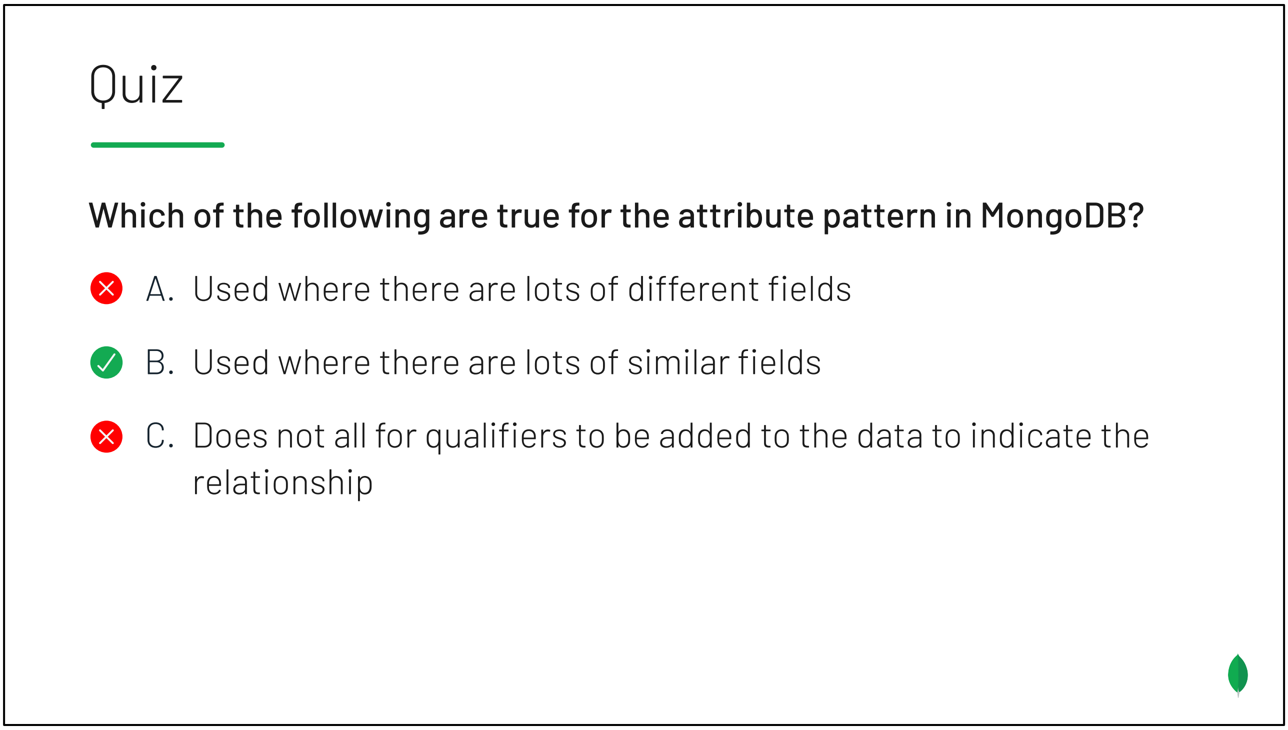
Quiz

Which of the following are true for the attribute pattern in MongoDB?
More than 1 answer choice can be correct.

○ A. Used where there are lots of different fields

○ B. Used where there are lots of similar fields

○ C. Does not all for qualifiers to be added to the data to indicate the relationship



Quiz

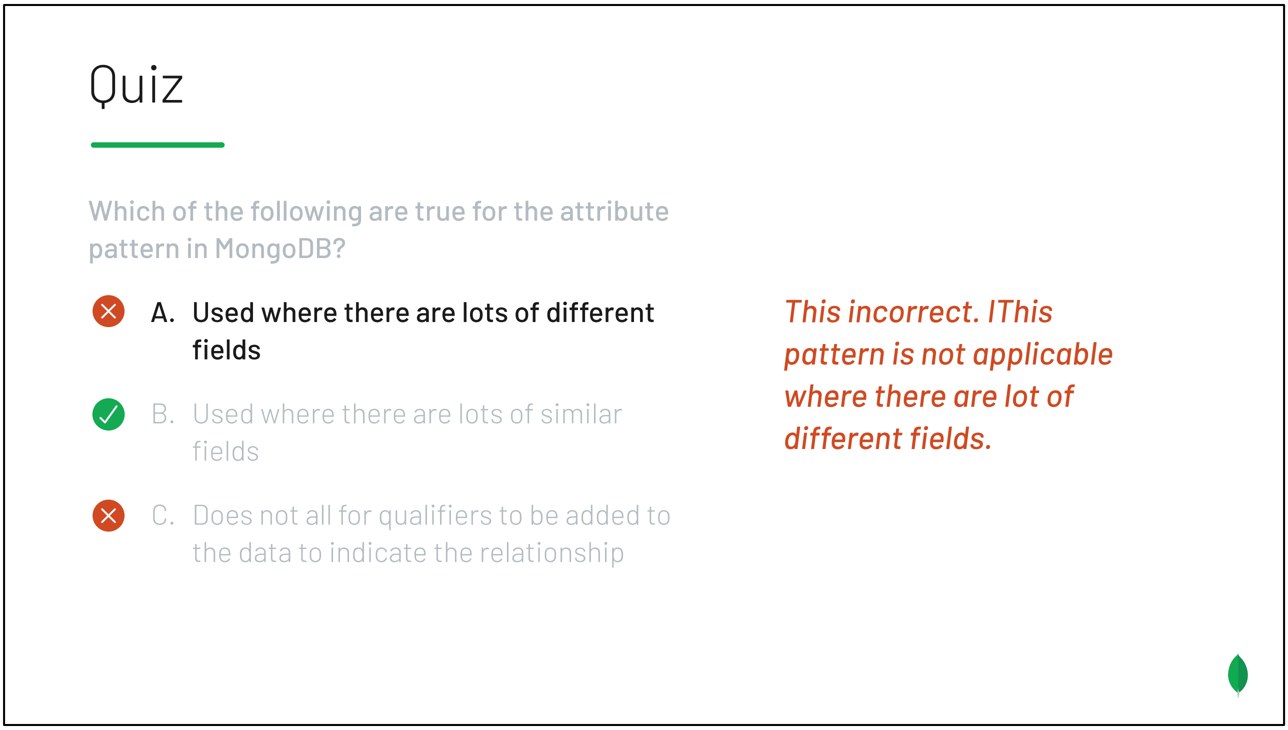
Which of the following are true for the attribute pattern in MongoDB?

✗ A. Used where there are lots of different fields

✓ B. Used where there are lots of similar fields

✗ C. Does not all for qualifiers to be added to the data to indicate the relationship

INCORRECT: Used where there are lots of different fields. - This is not correct for this pattern.
CORRECT: Used where there are lots of similar fields. - This pattern is ideal where there are lots of similar fields.
INCORRECT: Does not all for qualifiers to be added to the data to indicate the relationship. - This pattern does allow for qualifiers to be added to provide additional details about the relationship.



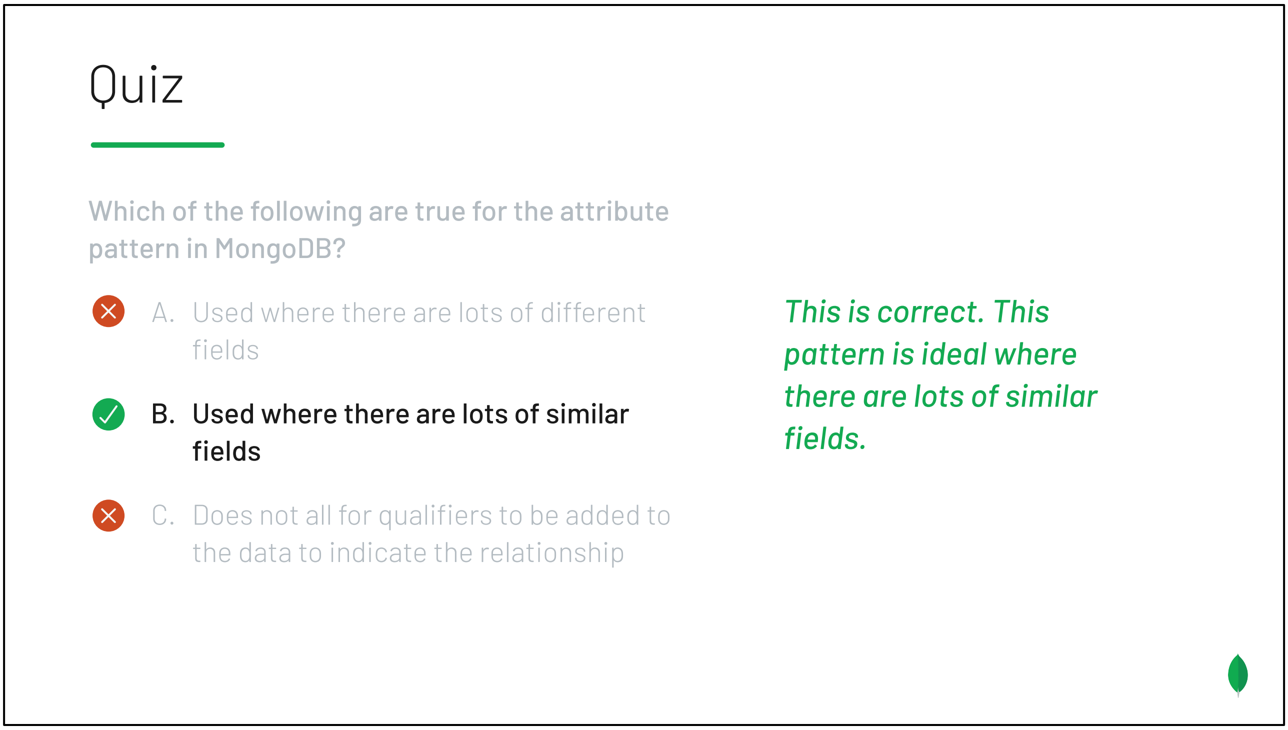
Quiz

Which of the following are true for the attribute pattern in MongoDB?

- ✗ A. Used where there are lots of different fields
- ✓ B. Used where there are lots of similar fields
- ✗ C. Does not all for qualifiers to be added to the data to indicate the relationship

This incorrect. IThis pattern is not applicable where there are lot of different fields.

INCORRECT: Used where there are lots of different fields. - This is incorrect. This pattern is not applicable where there are lot of different fields.



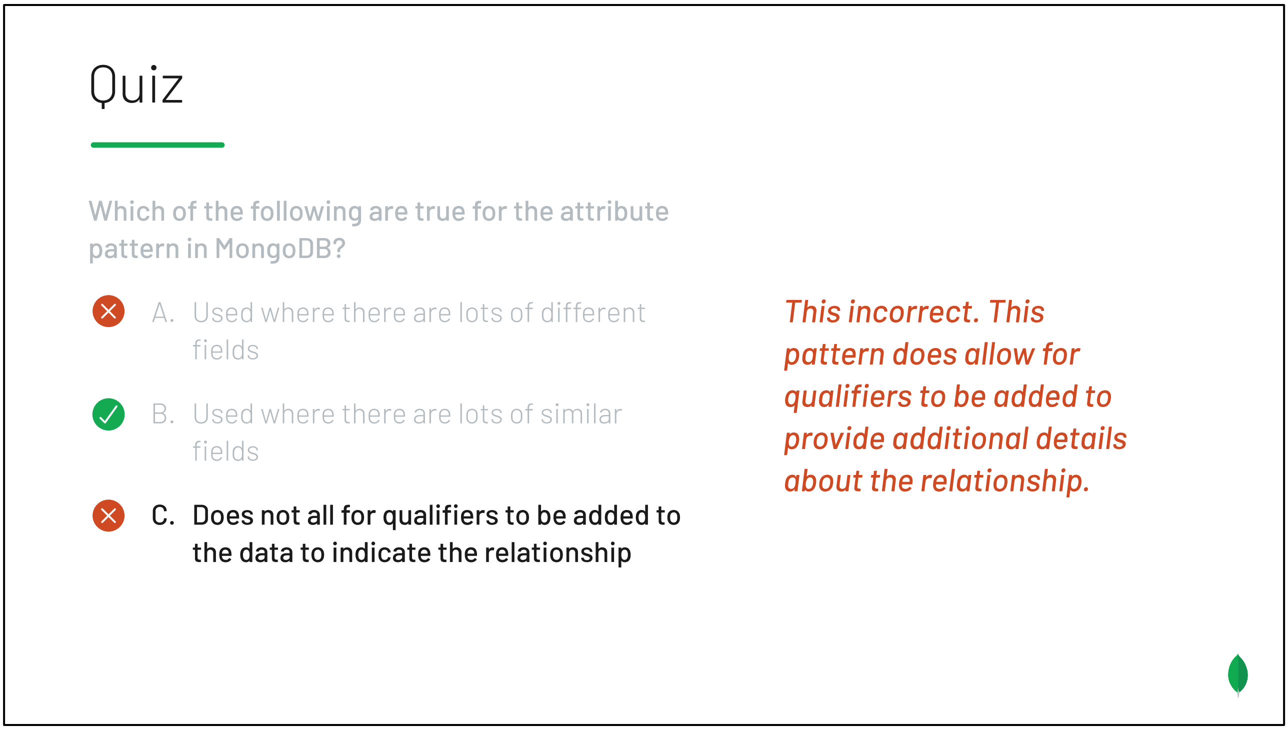
Quiz

Which of the following are true for the attribute pattern in MongoDB?

- ✗ A. Used where there are lots of different fields
- ✓ B. Used where there are lots of similar fields
- ✗ C. Does not all for qualifiers to be added to the data to indicate the relationship

This is correct. This pattern is ideal where there are lots of similar fields.

CORRECT: Used where there are lots of similar fields. - This is correct. This pattern is ideal where there are lots of similar fields.



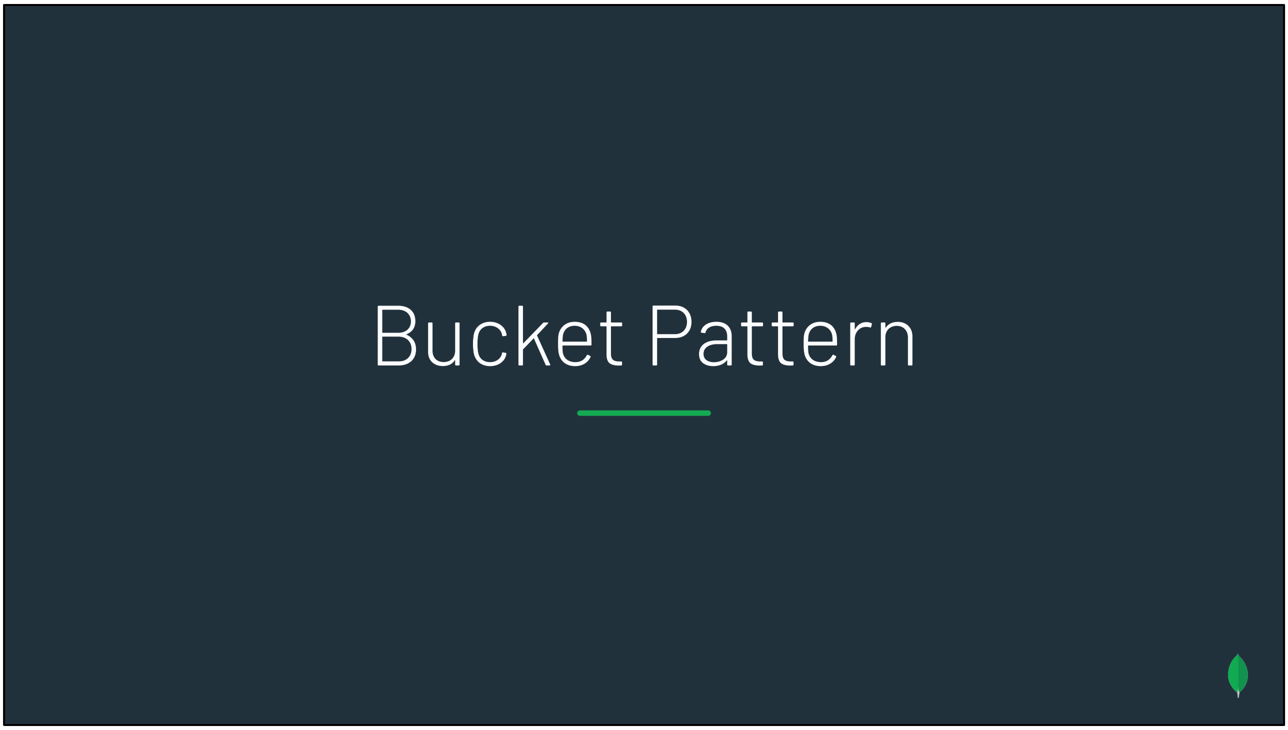
Quiz

Which of the following are true for the attribute pattern in MongoDB?

- ☒ A. Used where there are lots of different fields
- ✓ B. Used where there are lots of similar fields
- ☒ C. Does not all for qualifiers to be added to the data to indicate the relationship

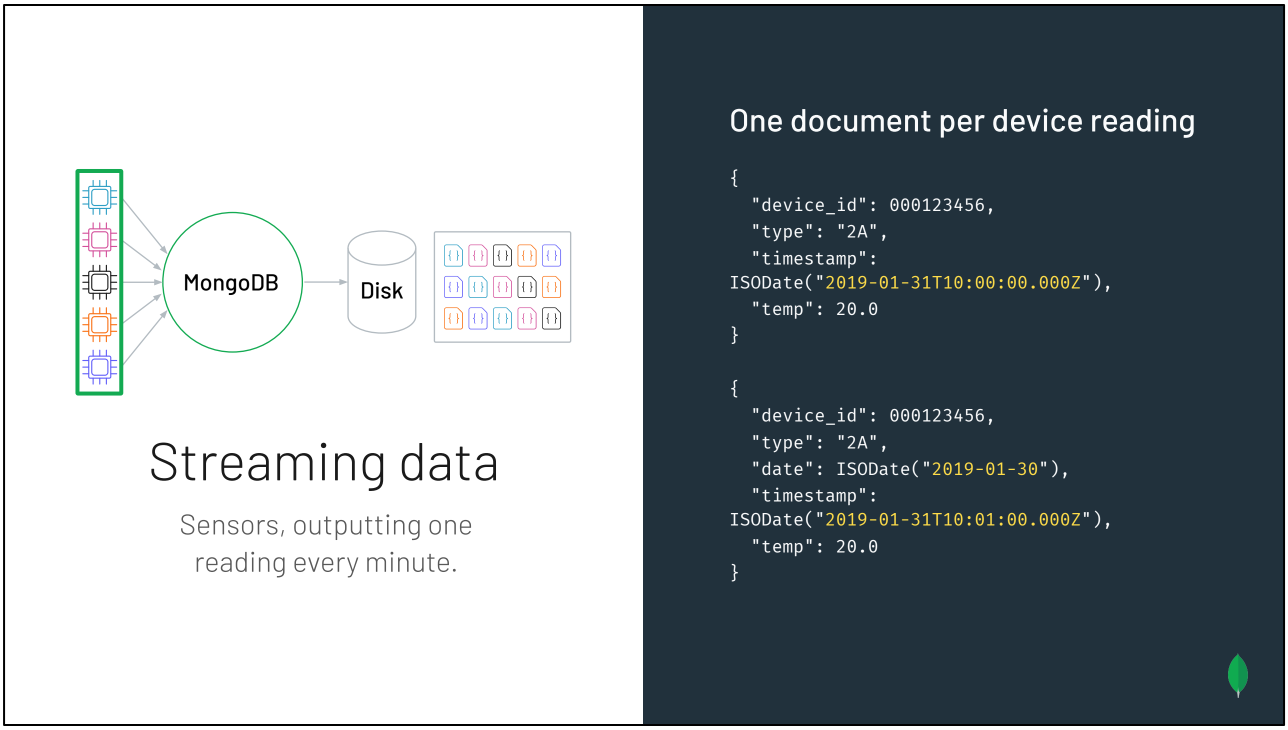
This incorrect. This pattern does allow for qualifiers to be added to provide additional details about the relationship.

INCORRECT: Does not all for qualifiers to be added to the data to indicate the relationship. - This is incorrect. This pattern does allow for qualifiers to be added to provide additional details about the relationship.



Bucket Pattern

Let’s look next at the Bucket Pattern which is really useful for streaming data, for instance where a lot of data is sent from sensors on a frequent basis.


MongoDB
Disk

Streaming data

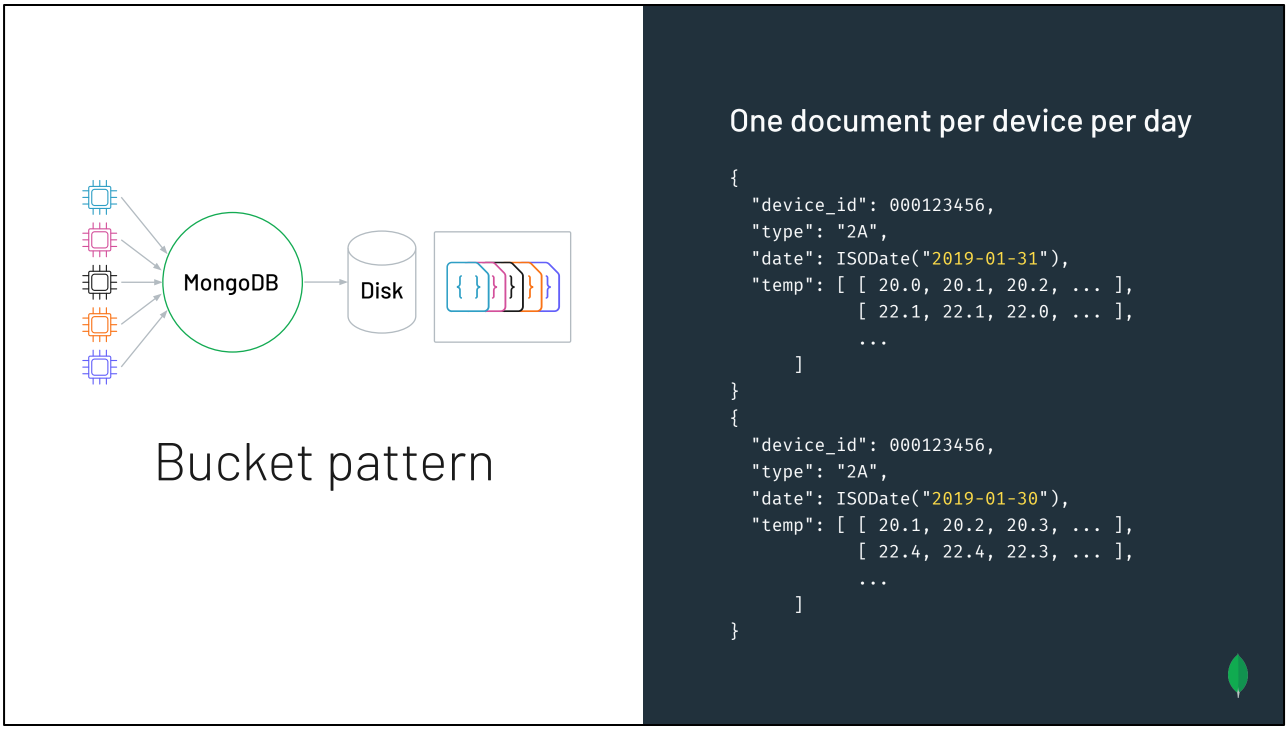
Sensors, outputting one
reading every minute.

One document per device reading

```
{
  "device_id": 000123456,
  "type": "2A",
  "timestamp":
ISODate("2019-01-31T10:00:00.000Z"),
  "temp": 20.0
}

{
  "device_id": 000123456,
  "type": "2A",
  "date": ISODate("2019-01-30"),
  "timestamp":
ISODate("2019-01-31T10:01:00.000Z"),
  "temp": 20.0
}
```

In the streaming data, like the internet of things use cases or similar use cases where a number of sources or sensors are generating data at a regular interval, say take a sensor and every minute. We could initially create a schema where each reading gets its own document.

This schema design will introduce a lot of overhead and additional reads/writes on the system and the number of sensors grow.



Bucket pattern

One document per device per day

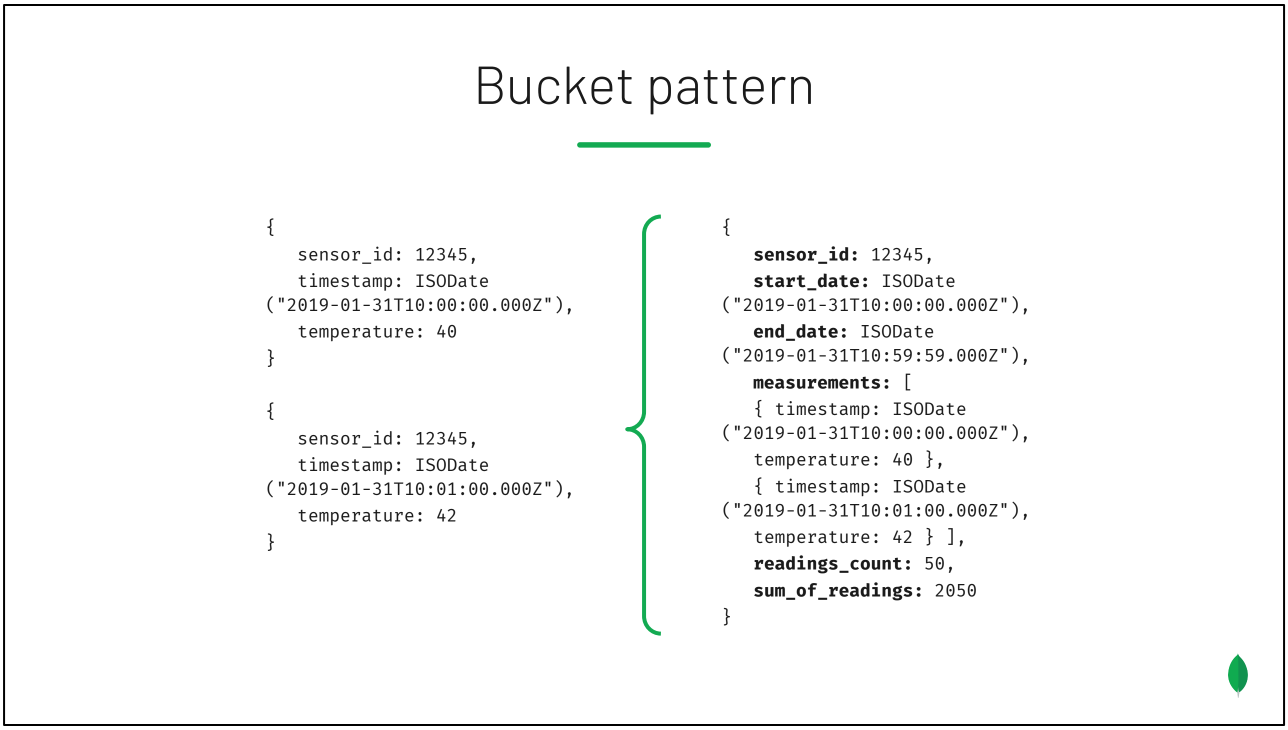
```
{
  "device_id": 000123456,
  "type": "2A",
  "date": ISODate("2019-01-31"),
  "temp": [ [ 20.0, 20.1, 20.2, ... ],
            [ 22.1, 22.1, 22.0, ... ],
            ...
      ]
}
{
  "device_id": 000123456,
  "type": "2A",
  "date": ISODate("2019-01-30"),
  "temp": [ [ 20.1, 20.2, 20.3, ... ],
            [ 22.4, 22.4, 22.3, ... ],
            ...
      ]
}
```

The bucket pattern is typically used in event type use cases, examples of this type of data stream includes sensors, financial trading data, or real time analytics.

Here's an example of a bucket pattern schema where instead of each document creating one document per recording, all of the recordings for a day are stored (bucketed) into a single document.

Let's look at this example for Internet of Things and sensors in more depth on the next slide.

When I send the slide deck out after class, you'll see a link to Patterns blog post on the Bucket Pattern https://developer.mongodb.com/how-to/bucket-pattern. Feel free to visit that link to learn more about the bucket pattern.



Bucket pattern

```
{
   sensor_id: 12345,
   timestamp: ISODate
("2019-01-31T10:00:00.000Z"),
   temperature: 40
}

{
   sensor_id: 12345,
   timestamp: ISODate
("2019-01-31T10:01:00.000Z"),
   temperature: 42
}
```

```
{
   sensor_id: 12345,
   start_date: ISODate
("2019-01-31T10:00:00.000Z"),
   end_date: ISODate
("2019-01-31T10:59:59.000Z"),
   measurements: [
   { timestamp: ISODate
("2019-01-31T10:00:00.000Z"),
   temperature: 40 },
   { timestamp: ISODate
("2019-01-31T10:01:00.000Z"),
   temperature: 42 } ],
   readings_count: 50,
   sum_of_readings: 2050
}
```

The bucket pattern is typically used in event type use cases, examples of this type of data stream includes sensors, financial trading data, or real time analytics.

A first design of the schema (shown on the left of the slide) may use a document to store the sensor_id, the timestamp for a recording, and the value of the recording (temperature in this case).

The bucket pattern refactors the earlier schema to hold a range or bucket of measurements, in this example it’s an hour of sensor recordings for a single sensor.

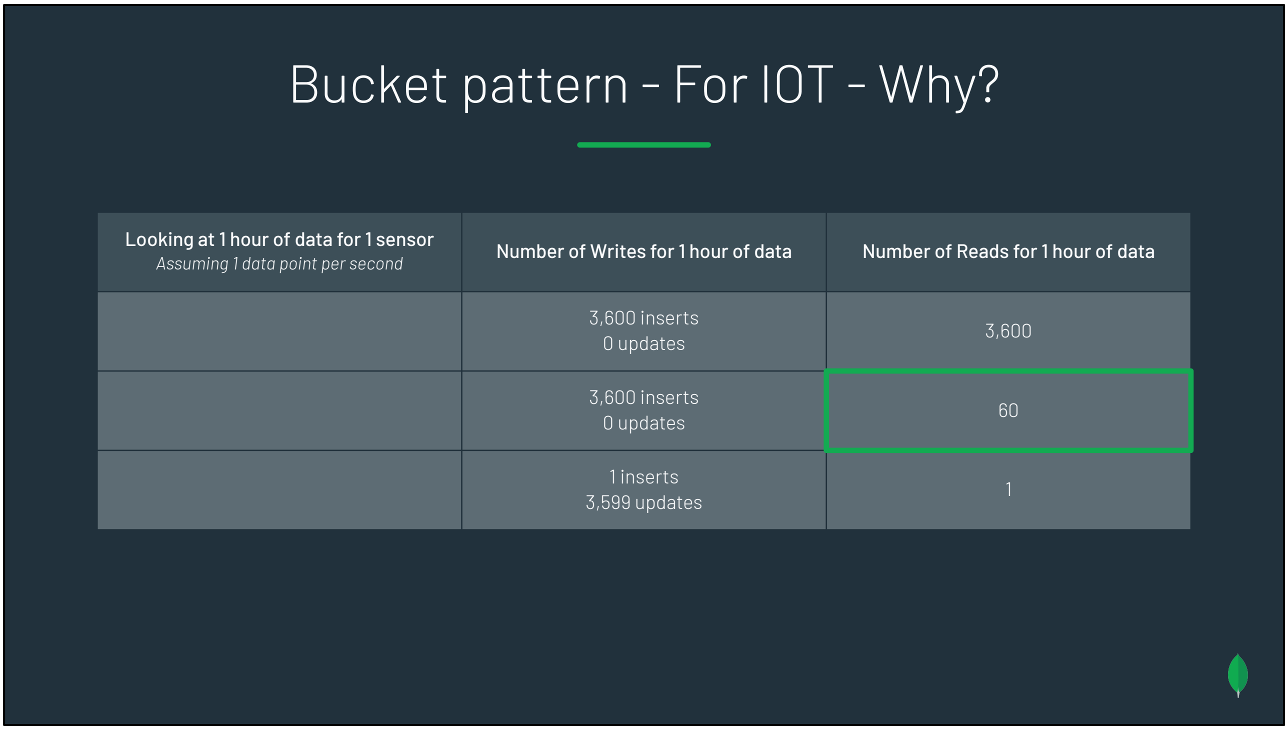
In this example, we’ll assume there are still a number of measurements to be added which would be added to the measurements array. In addition, we would also increment the readings_count and update the sum_of_readings.

Using a bucket pattern allows applications to easily pull up a particular bucket and determine the average reading (sum/count). In this example of sensors and the wider time series data, a question is often what was the average reading during a period for a date and indeed we could easily add location as a field to our document above to allow us to answer where as well.

Bucketing and pre-aggregation allow us to easily answer those types of questions.

Specifically, this pattern reduces the overall number of documents in a collection, improves the indexing performance, and if pre-aggregation is used, it can simplify queries and data access.

When I send the slide deck out after class, you'll see a link to a HowTo blog post on using the Bucket Pattern https://developer.mongodb.com/how-to/bucket-pattern. Feel free to visit that link to learn more about the bucket pattern.



Bucket pattern - For IOT - Why?

Looking at 1 hour of data for 1 sensor *Assuming 1 data point per second*	Number of Writes for 1 hour of data	Number of Reads for 1 hour of data
	3,600 inserts 0 updates	3,600
	3,600 inserts 0 updates	60
	1 inserts 3,599 updates	1

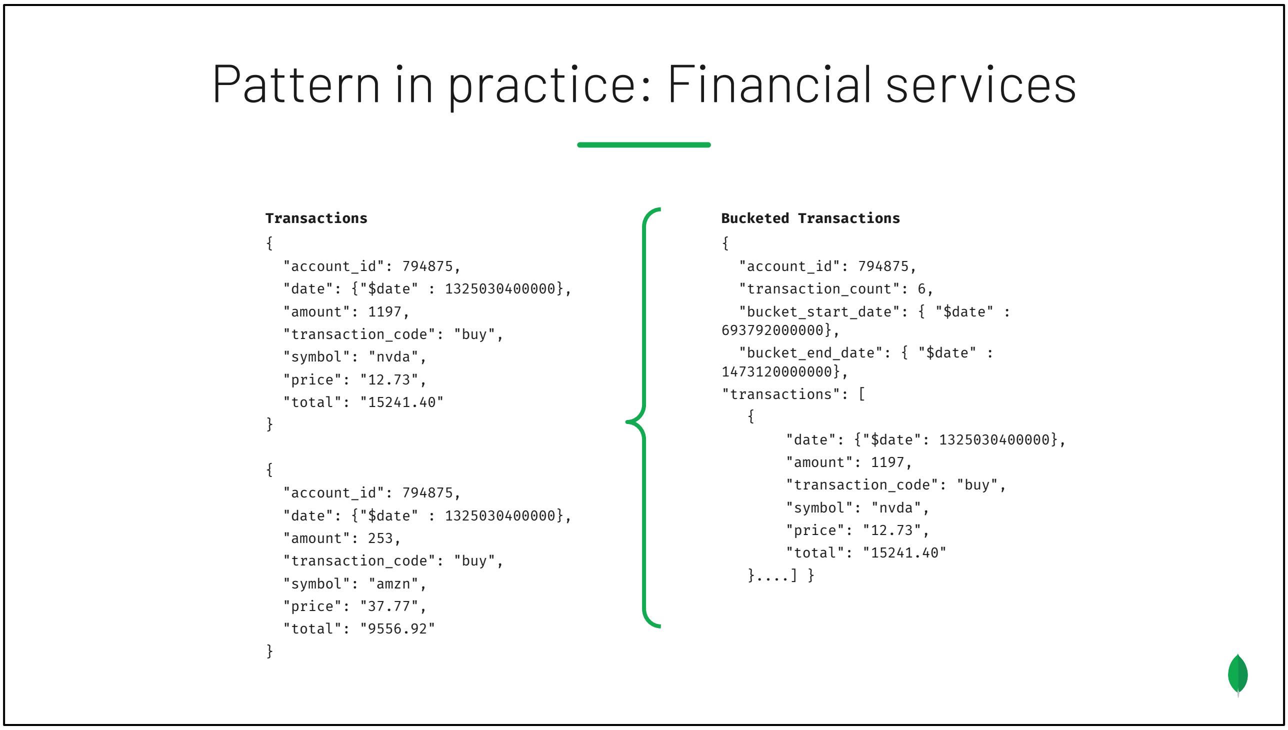
We're looked at the bucket pattern for sensor data, let's take a practice example of the impact this might have given one sensor that records data every second and let's look at how many operations are need to write that data and also to read back the full hours data.

If it takes 3,600 reads to retrieve the unbucketed schema, how many reads does it take with the bucketed to one minute schema ?

The answer is 60.

At bucketing per minute you will have 60x less documents, and hence 60x less index entries, which then in turn reduce your memory requirements to a much lower requirement.

It's also possible to bucket the bucket, in this case we could have each minute's worth of sensor data is an array entry into a document holding one hour's worth of sensor data.



Pattern in practice: Financial services

Transactions

```
{
  "account_id": 794875,
  "date": {"$date" : 1325030400000},
  "amount": 1197,
  "transaction_code": "buy",
  "symbol": "nvda",
  "price": "12.73",
  "total": "15241.40"
}

{
  "account_id": 794875,
  "date": {"$date" : 1325030400000},
  "amount": 253,
  "transaction_code": "buy",
  "symbol": "amzn",
  "price": "37.77",
  "total": "9556.92"
}
```

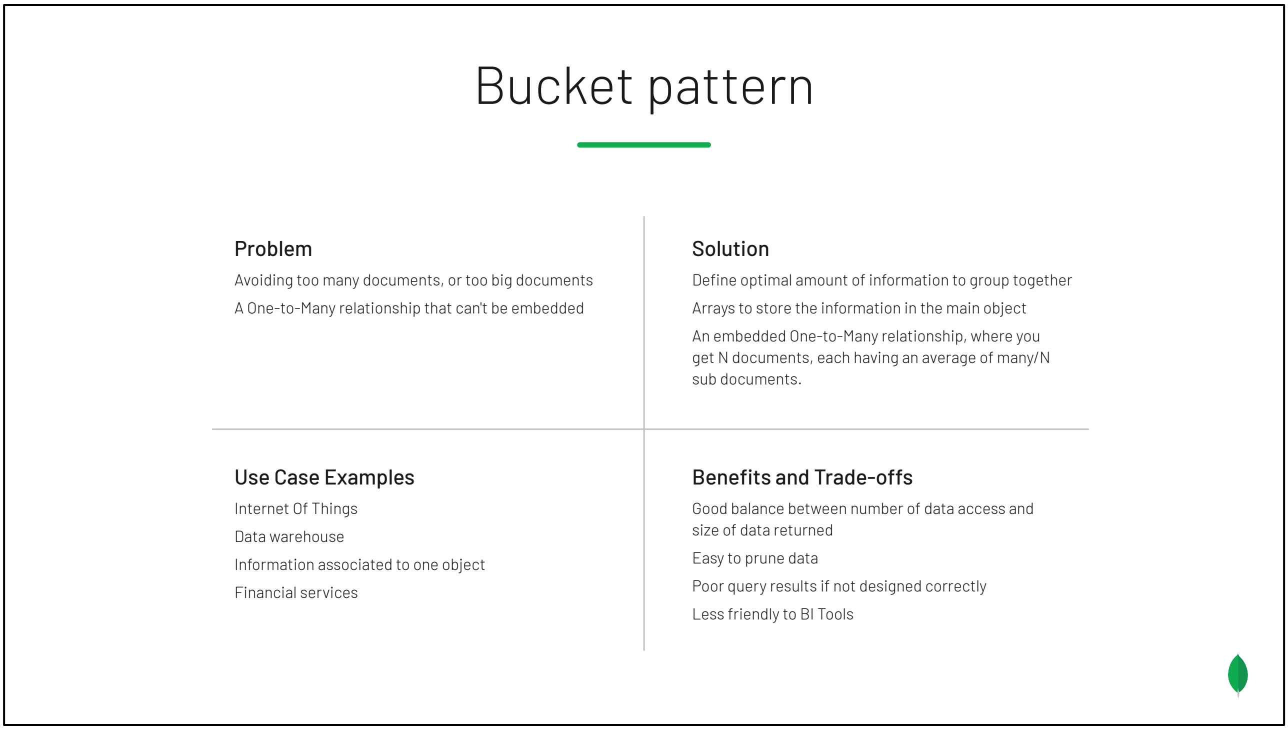
Bucketed Transactions

```
{
  "account_id": 794875,
  "transaction_count": 6,
  "bucket_start_date": { "$date" :
693792000000},
  "bucket_end_date": { "$date" :
1473120000000},
"transactions": [
   {
        "date": {"$date": 1325030400000},
        "amount": 1197,
        "transaction_code": "buy",
        "symbol": "nvda",
        "price": "12.73",
        "total": "15241.40"
   }....] }
```

Another example of this pattern in practice can be seen in the Atlas Sample Dataset, specifically in the sample analytics data. It provides three collections to represent customers, accounts, and transactions.

In the case of the transactions collection, it uses the bucket pattern as shown in the schema on the right of the slide.

When I send the slide deck out after class, you'll see a link to a MongoDB documentation page on using the Sample Analytics dataset https://docs.atlas.mongodb.com/sample-data/sample-analytics/. Feel free to visit that link to learn more about the sample analytics dataset.



Bucket pattern

Problem

Avoiding too many documents, or too big documents

A One-to-Many relationship that can't be embedded

Solution

Define optimal amount of information to group together

Arrays to store the information in the main object

An embedded One-to-Many relationship, where you get N documents, each having an average of many/N sub documents.

Use Case Examples

Internet Of Things

Data warehouse

Information associated to one object

Financial services

Benefits and Trade-offs

Good balance between number of data access and size of data returned

Easy to prune data

Poor query results if not designed correctly

Less friendly to BI Tools

As noted previously we'll summarise each of pattern under the same four headings of Problem, Solution, Use Case Examples, and Benefits & Trade-offs.

In terms of the problem the bucket problem is seeking to resolve it is concerned with too many documents or documents that are too big. We looked at the IOT area and highlighted how one document per sensor reading might be inefficient when compared to bucketing the readings. The bucket pattern is also helpful where it is a one-to-many relationship that cannot be embedded.

In terms of the solution, the bucket pattern allows for the optimal amount of information to be grouped together. This might be sensor data at a one minute bucket or at a bucket of buckets where this data is bucketed at the second level within a minute bucket to give an hour document of sensor readings. Arrays can be used in the main object to implement this pattern. This pattern highlights how you can embed one-to-many relationships with n documents each having many other sub documents.

The use case we already covered was sensor data in the Internet of Things but this pattern is applicable in data warehouses, financial services or where a single object may have lots of information associated to it.

The bucket pattern provides a good balance between data access and data size returned when accessed. It is also easy to prune data when using this pattern given both the clear structure of how it's stored in a document or at a document level using the likes of a TTL index to remove data at a certain time or after a certain time.

The bucket pattern can lead to poor queries if it is not designed carefully and it is not well matched with current BI tools for visualisation/analytics.

When I send the slide deck out after class, you'll see a link to a HowTo blog post on using the Bucket Pattern https://developer.mongodb.com/how-to/bucket-pattern. Feel free to visit that link to learn more about the bucket pattern.

Quiz

Quiz

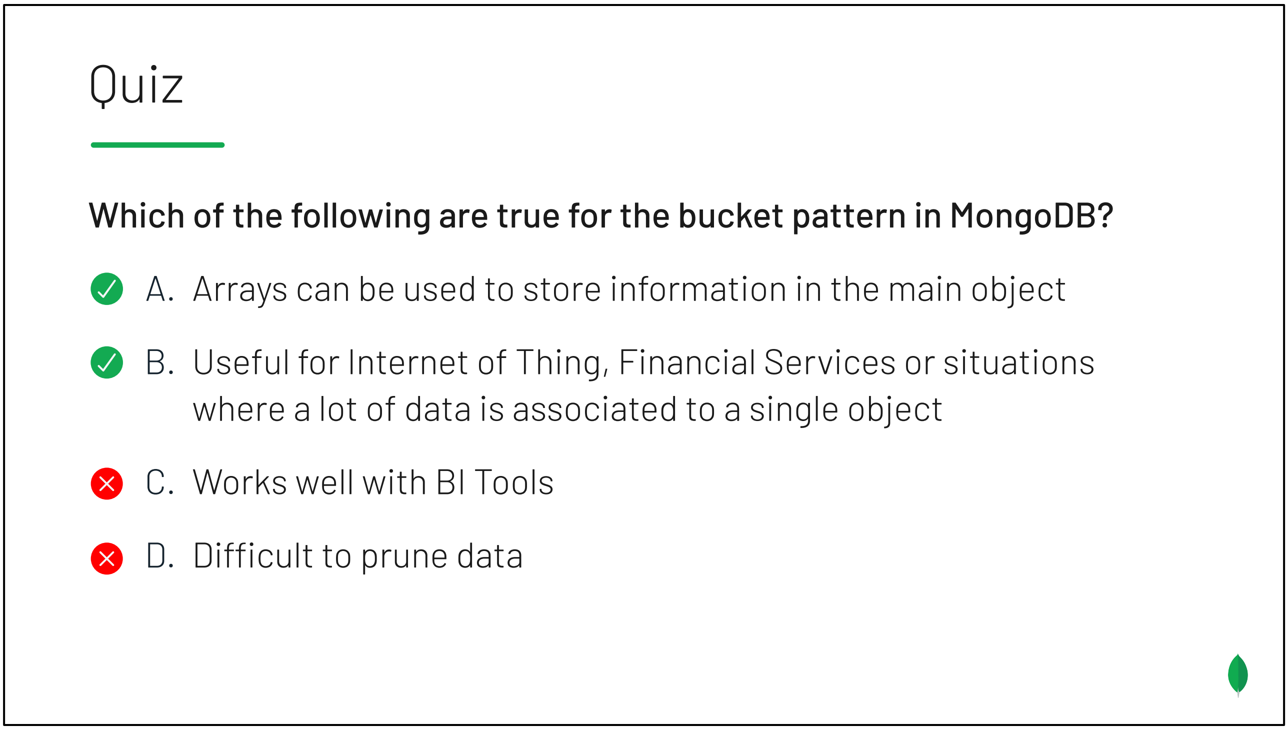
Which of the following are true for the bucket pattern in MongoDB?
More than 1 answer choice can be correct.

● A. Arrays can be used to store information in the main object

● B. Useful for Internet of Thing, Financial Services or situations where a lot of data is associated to a single object

● C. Works well with BI Tools

● D. Difficult to prune data



Quiz

Which of the following are true for the bucket pattern in MongoDB?

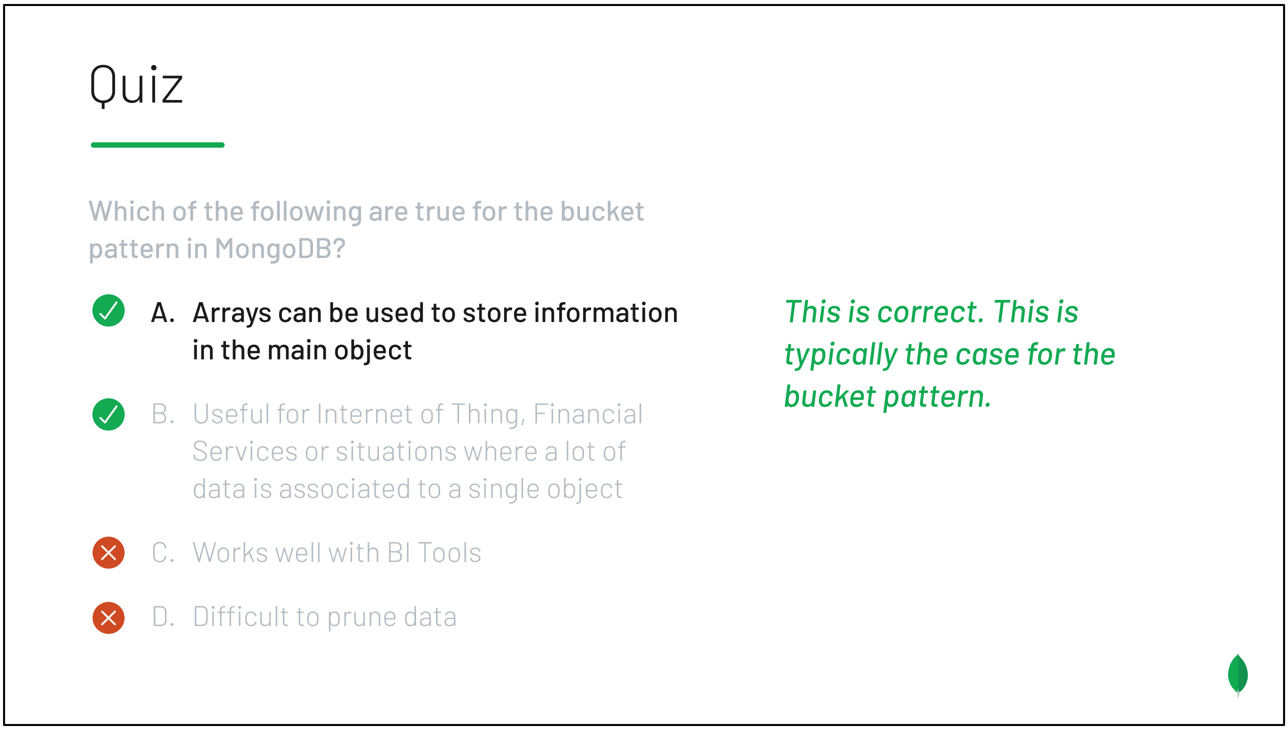
✓ A. Arrays can be used to store information in the main object

✓ B. Useful for Internet of Thing, Financial Services or situations where a lot of data is associated to a single object

✗ C. Works well with BI Tools

✗ D. Difficult to prune data

CORRECT: Arrays can be used to store information in the main object. - This is typically the case for the bucket pattern.
CORRECT: Useful for Internet of Thing, Financial Services or situations where a lot of data is associated to a single object. - It can be used in other use cases, these are only an example where it’s clear that the pattern is known to be useful.
INCORRECT: Works well with BI Tools. - This is incorrect as due to the formatting of the data and how it is structured most BI tools are not well suited to visualising it.
INCORRECT: Difficult to prune data. - This is incorrect due to the structuring of the data within the document and at the document level, this pattern makes it easy to prune data.



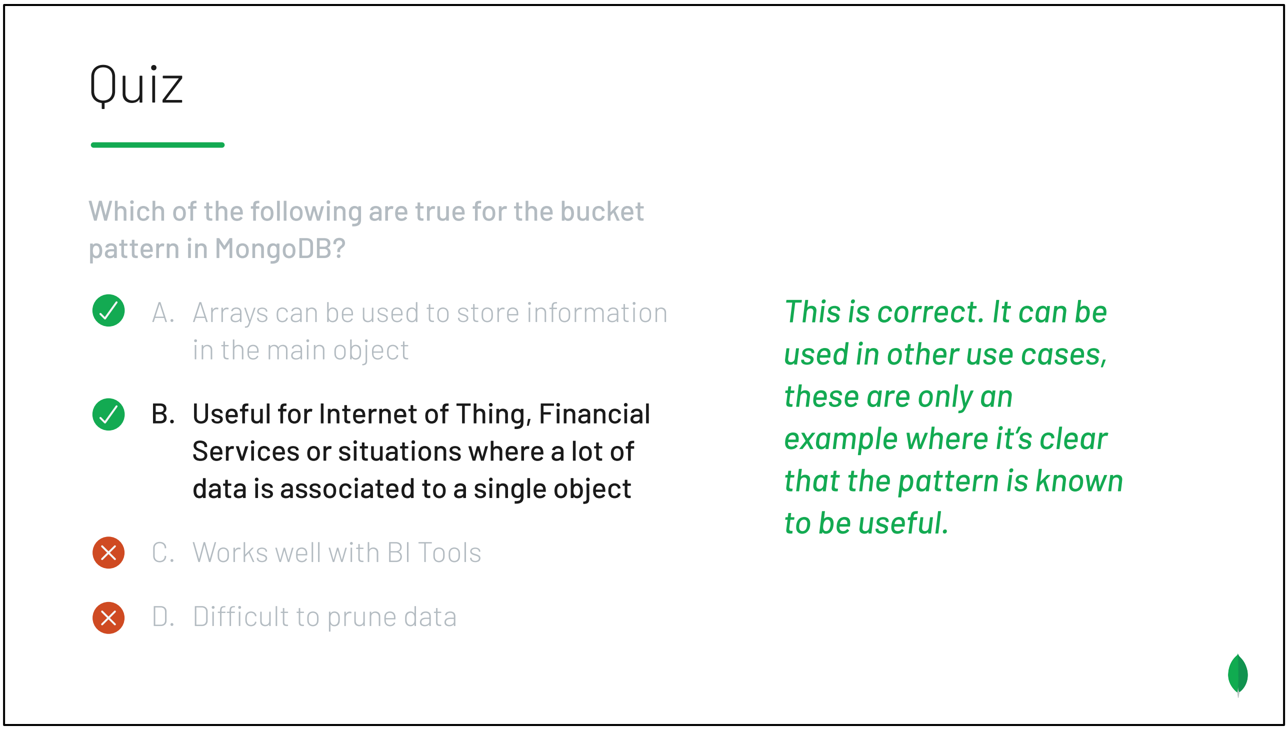
Quiz

Which of the following are true for the bucket pattern in MongoDB?

- ✓ A. **Arrays can be used to store information in the main object**
- ✓ B. Useful for Internet of Thing, Financial Services or situations where a lot of data is associated to a single object
- ✗ C. Works well with BI Tools
- ✗ D. Difficult to prune data

This is correct. This is typically the case for the bucket pattern.

CORRECT: Arrays can be used to store information in the main object. - This is correct. This is typically the case for the bucket pattern.



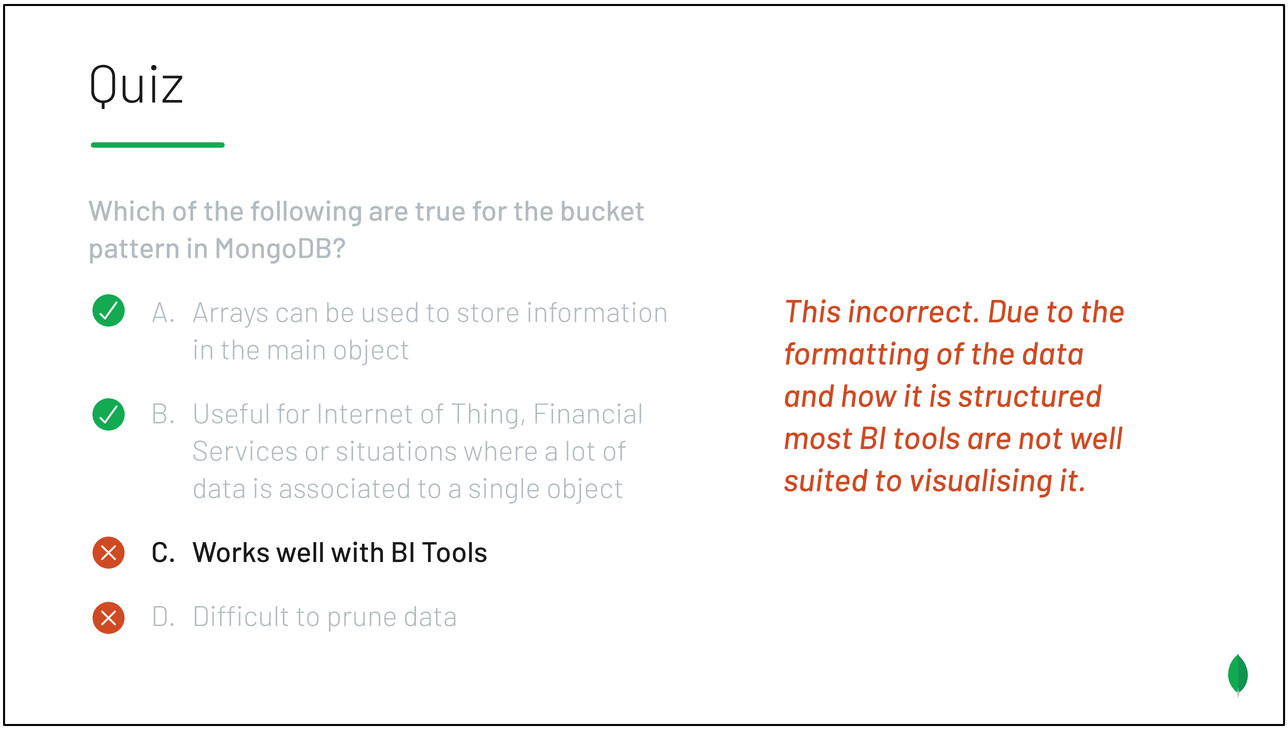
Quiz

Which of the following are true for the bucket pattern in MongoDB?

- ✓ A. Arrays can be used to store information in the main object
- ✓ B. Useful for Internet of Thing, Financial Services or situations where a lot of data is associated to a single object
- ✗ C. Works well with BI Tools
- ✗ D. Difficult to prune data

This is correct. It can be used in other use cases, these are only an example where it's clear that the pattern is known to be useful.

CORRECT: Useful for Internet of Thing, Financial Services or situations where a lot of data is associated to a single object. - This is correct. It can be used in other use cases, these are only an example where it’s clear that the pattern is known to be useful.



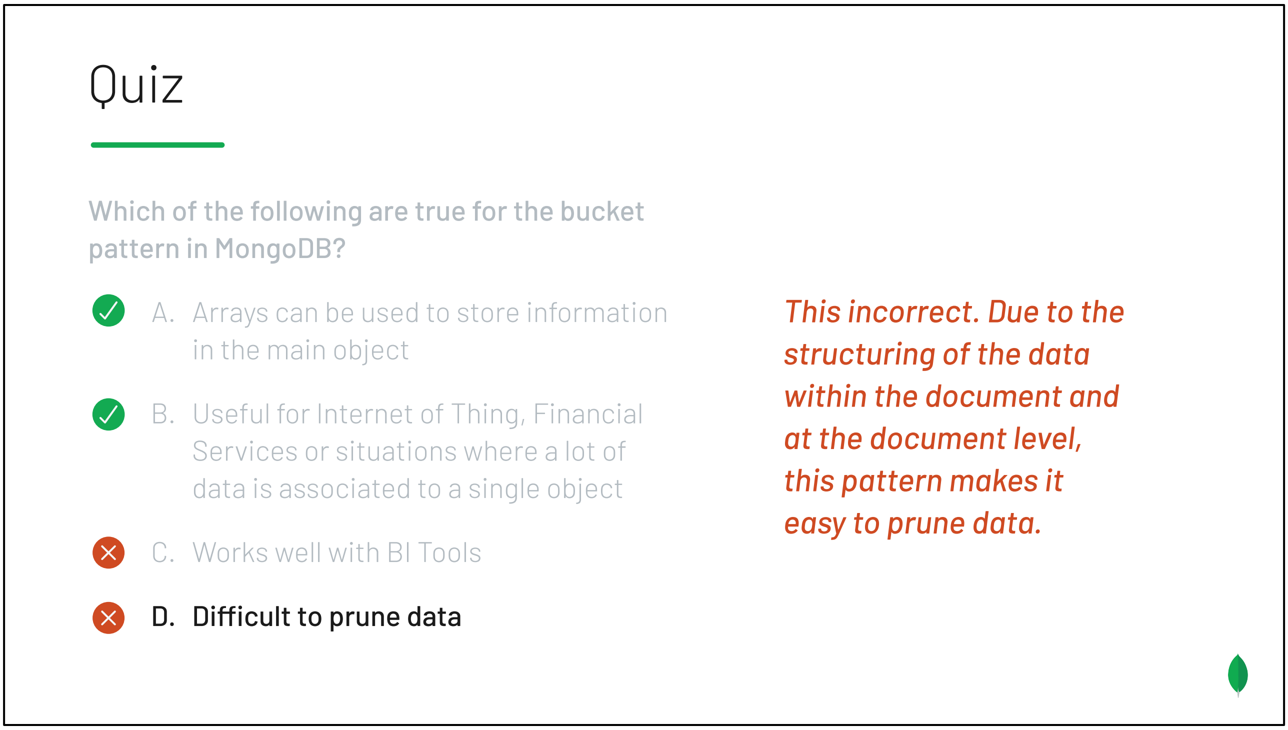
Quiz

Which of the following are true for the bucket pattern in MongoDB?

- ✓ A. Arrays can be used to store information in the main object
- ✓ B. Useful for Internet of Thing, Financial Services or situations where a lot of data is associated to a single object
- ✗ C. Works well with BI Tools
- ✗ D. Difficult to prune data

This incorrect. Due to the formatting of the data and how it is structured most BI tools are not well suited to visualising it.

INCORRECT: Works well with BI Tools. - This is incorrect. Due to the formatting of the data and how it is structured most BI tools are not well suited to visualising it.



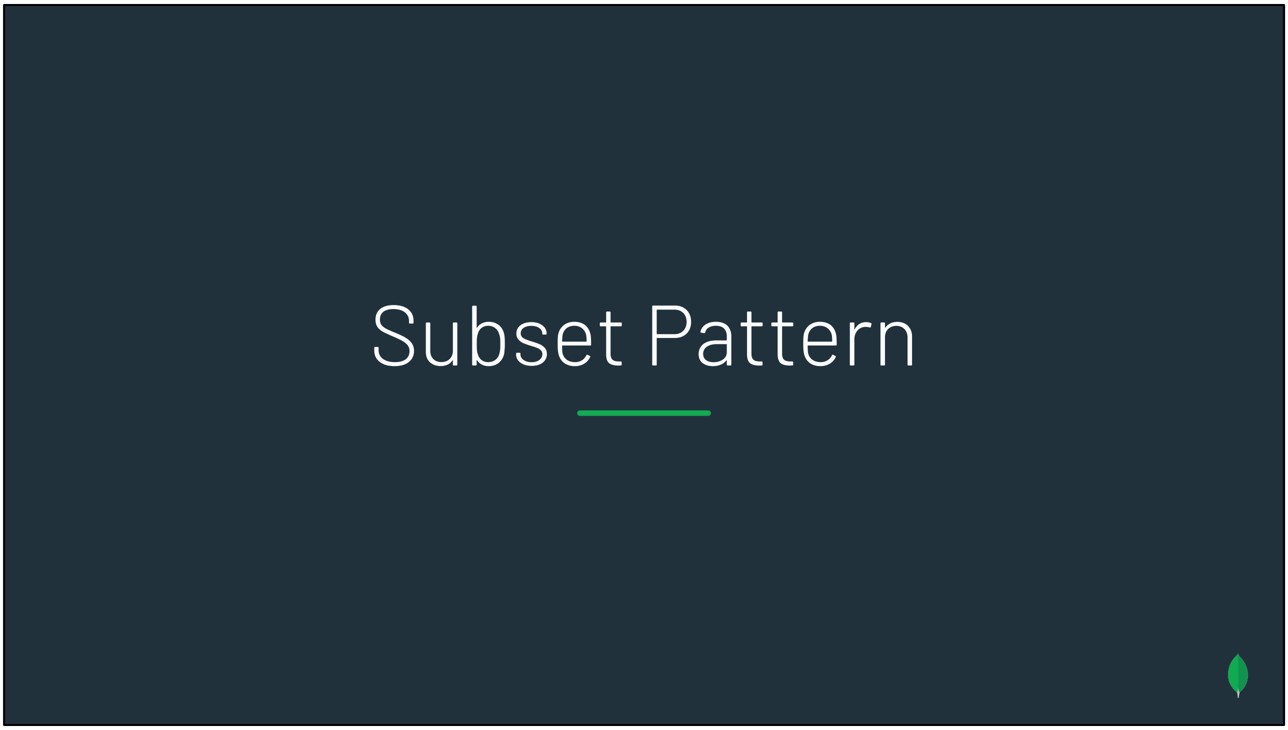
Quiz

Which of the following are true for the bucket pattern in MongoDB?

- ✓ A. Arrays can be used to store information in the main object
- ✓ B. Useful for Internet of Thing, Financial Services or situations where a lot of data is associated to a single object
- ✗ C. Works well with BI Tools
- ✗ **D. Difficult to prune data**

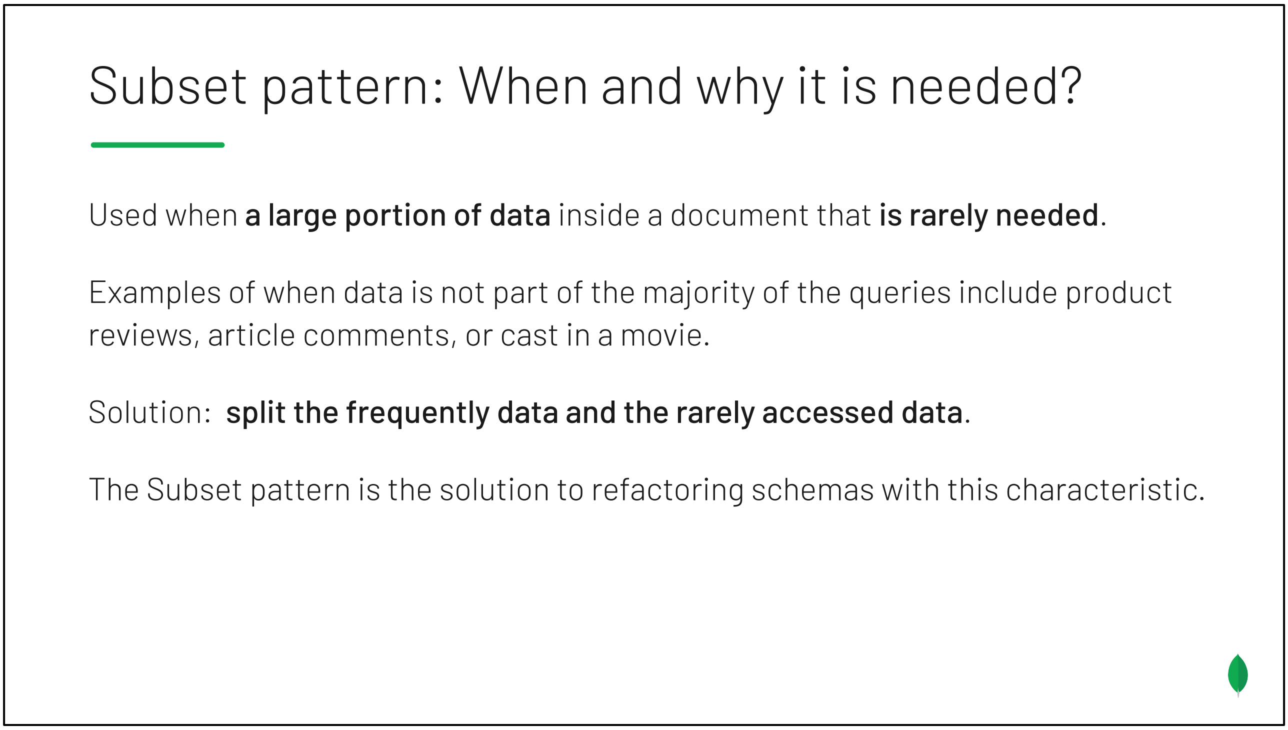
This incorrect. Due to the structuring of the data within the document and at the document level, this pattern makes it easy to prune data.

INCORRECT: Difficult to prune data. - This is incorrect. Due to the structuring of the data within the document and at the document level, this pattern makes it easy to prune data.



Subset Pattern

Let’s look at our last pattern in this lesson, the Subset Pattern.



Subset pattern: When and why it is needed?

Used when **a large portion of data** inside a document that **is rarely needed**.

Examples of when data is not part of the majority of the queries include product reviews, article comments, or cast in a movie.

Solution: **split the frequently data and the rarely accessed data**.

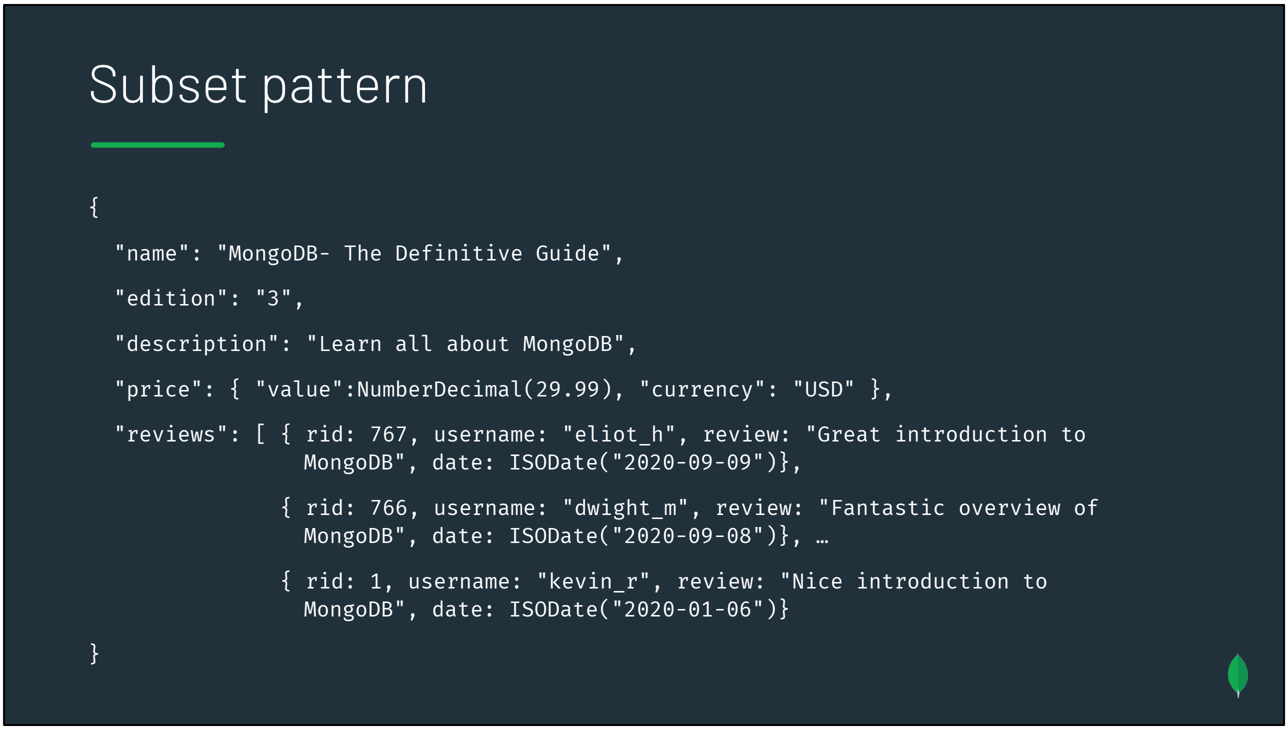
The Subset pattern is the solution to refactoring schemas with this characteristic.

The Subset Pattern is typically used in the case where there is:

- A large portion of data within a document that is rarely accessed/needed

An example of this, is in terms of product reviews, article comments, or actors in a movie. In the case of product reviews, it’s typically to shown the last ten or twenty for a specific item but beyond these the rest of the reviews are infrequently requested/queried.

The Subset patten splits out the frequently accessed data and the rarely accessed data. It refactors the schema to support a smaller main object and pushes the rarely accessed data from that document to a new document in another collection.



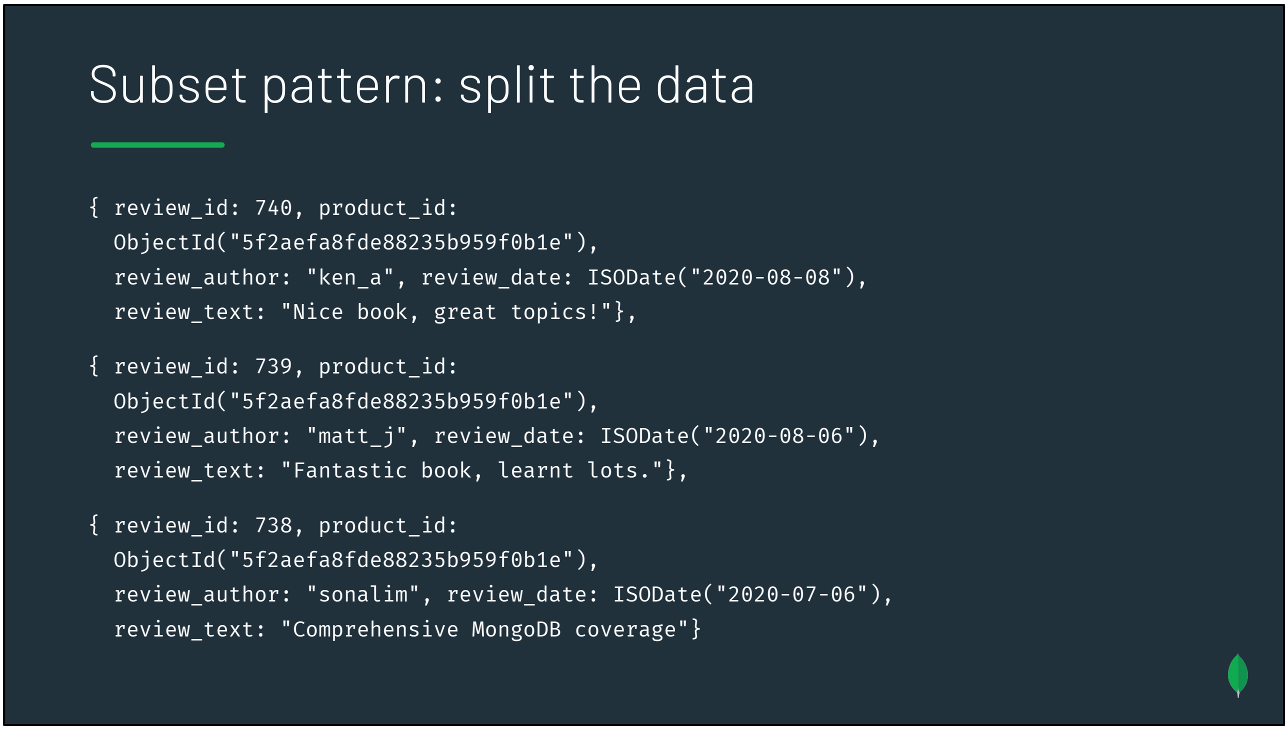
Subset pattern

```
{
  "name": "MongoDB- The Definitive Guide",
  "edition": "3",
  "description": "Learn all about MongoDB",
  "price": { "value":NumberDecimal(29.99), "currency": "USD" },
  "reviews": [ { rid: 767, username: "eliot_h", review: "Great introduction to
                 MongoDB", date: ISODate("2020-09-09")},
               { rid: 766, username: "dwight_m", review: "Fantastic overview of
                 MongoDB", date: ISODate("2020-09-08")}, …
               { rid: 1, username: "kevin_r", review: "Nice introduction to
                 MongoDB", date: ISODate("2020-01-06")}
}
```

Here's a example of a review document for an ecommerce catalogue which sells books.
In the initial schema all of the reviews are stored in an array in the product document.

This can lead to retrieving a large amount of data that isn't typically required by queries.
The Subset pattern splits this data, in this example the last ten reviews might be kept within the product document.

When I send the slide deck out after class, you'll see a link to Patterns blog post on the Subset Pattern https://developer.mongodb.com/how-to/subset-pattern. Feel free to visit that link to learn more about the subset pattern.



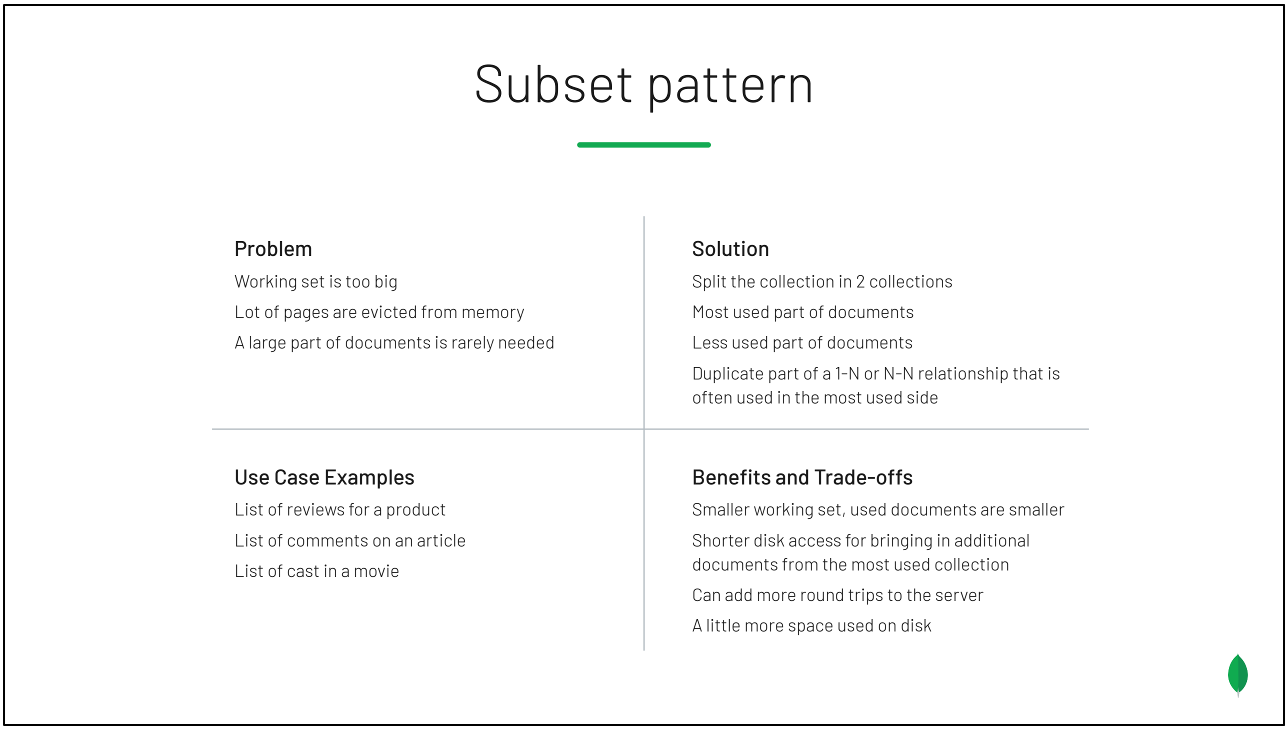
Subset pattern: split the data

```
{ review_id: 740, product_id:
  ObjectId("5f2aefa8fde88235b959f0b1e"),
  review_author: "ken_a", review_date: ISODate("2020-08-08"),
  review_text: "Nice book, great topics!"},

{ review_id: 739, product_id:
  ObjectId("5f2aefa8fde88235b959f0b1e"),
  review_author: "matt_j", review_date: ISODate("2020-08-06"),
  review_text: "Fantastic book, learnt lots."},

{ review_id: 738, product_id:
  ObjectId("5f2aefa8fde88235b959f0b1e"),
  review_author: "sonalim", review_date: ISODate("2020-07-06"),
  review_text: "Comprehensive MongoDB coverage"}
```

The remaining reviews (outside of the latest / last ten) are moved to their own dedicated collection, say reviews.

In using the Subset pattern, the main consideration is where to split your data, specifically what part of the frequently used data should stay in the original/”main” collection and which part should be moved to a new less frequently accessed collection. In the case of product reviews, it might be ten or twenty is around the point we want to split the data.

The tradeoff with this pattern is that additional queries will be required if we need to pull more than the ten or twenty reviews for a product.



Subset pattern

Problem

Working set is too big

Lot of pages are evicted from memory

A large part of documents is rarely needed

Solution

Split the collection in 2 collections

Most used part of documents

Less used part of documents

Duplicate part of a 1-N or N-N relationship that is often used in the most used side

Use Case Examples

List of reviews for a product

List of comments on an article

List of cast in a movie

Benefits and Trade-offs

Smaller working set, used documents are smaller

Shorter disk access for bringing in additional documents from the most used collection

Can add more round trips to the server

A little more space used on disk

As noted previously we'll summarise each of pattern under the same four headings of Problem, Solution, Use Case Examples, and Benefits & Trade-offs.

The Subset pattern is concerned in situations where the working set is too large and large documents are a significant factor in the contributing to the workset. The large documents can cause lots of pages to be evicted from memory and typically only a subset of the information in any particular document is only being used.

The solution to this situation is to split the collection into two different parts, one holding the most frequently used parts of the documents and a second holding the less frequently used/accessed parts of the document. This is done by locating a 1-N or N-N relationship on the most used side and breaking on this.

In terms of use, we've highlighted the product reviews but the same rationale applies for comments on an article or for cast in a movie.

Splitting the collection provides a much smaller working set as it reduces the size of documents and allows more of these used documents to be kept in memory. This approach can add some more query round trips to the server if the workload changes and the less used parts of the documents are required for queries and it does add a little more disk space usage.

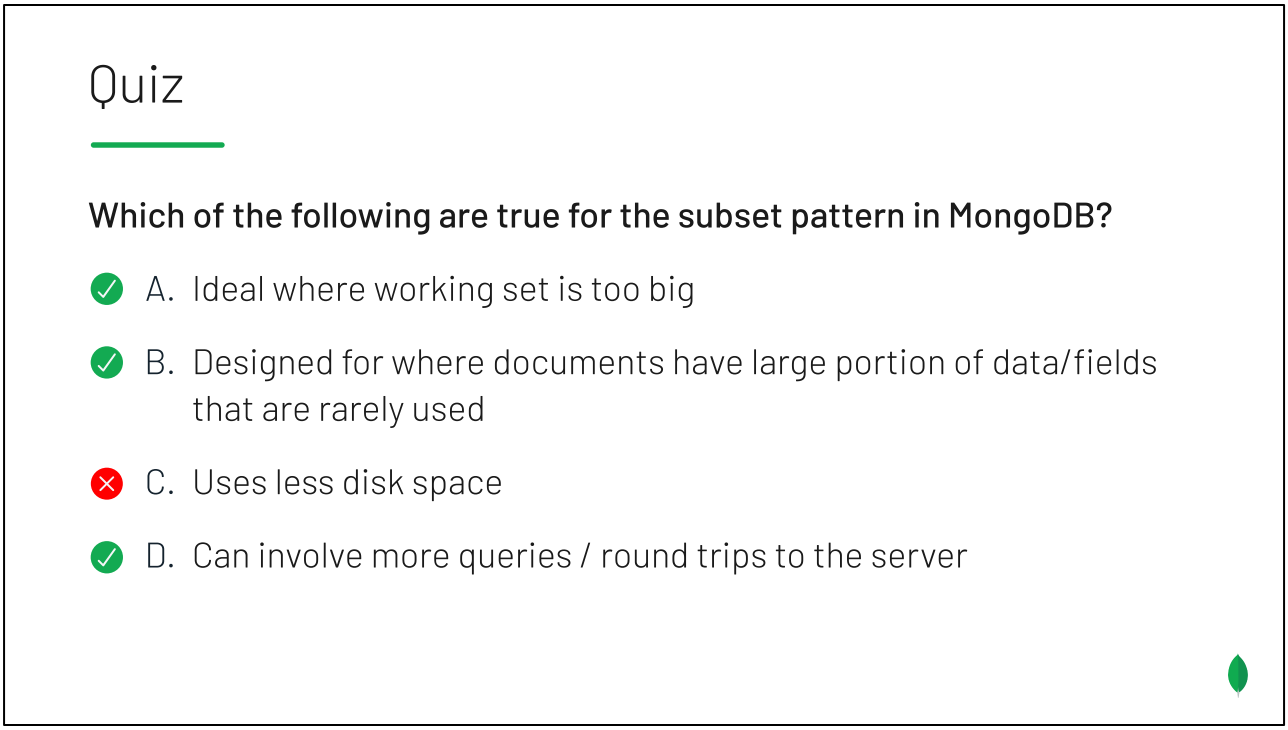
When I send the slide deck out after class, you'll see a link to Patterns blog post on the Subset Pattern https://developer.mongodb.com/how-to/subset-pattern. Feel free to visit that link to learn more about the subset pattern.

Quiz

Quiz

Which of the following are true for the subset pattern in MongoDB?
More than 1 answer choice can be correct.

- ● A. Ideal where working set is too big
- ● B. Designed for where documents have large portion of data/fields that are rarely used
- ● C. Uses less disk space
- ● D. Can involve more queries / round trips to the server



Quiz

Which of the following are true for the subset pattern in MongoDB?

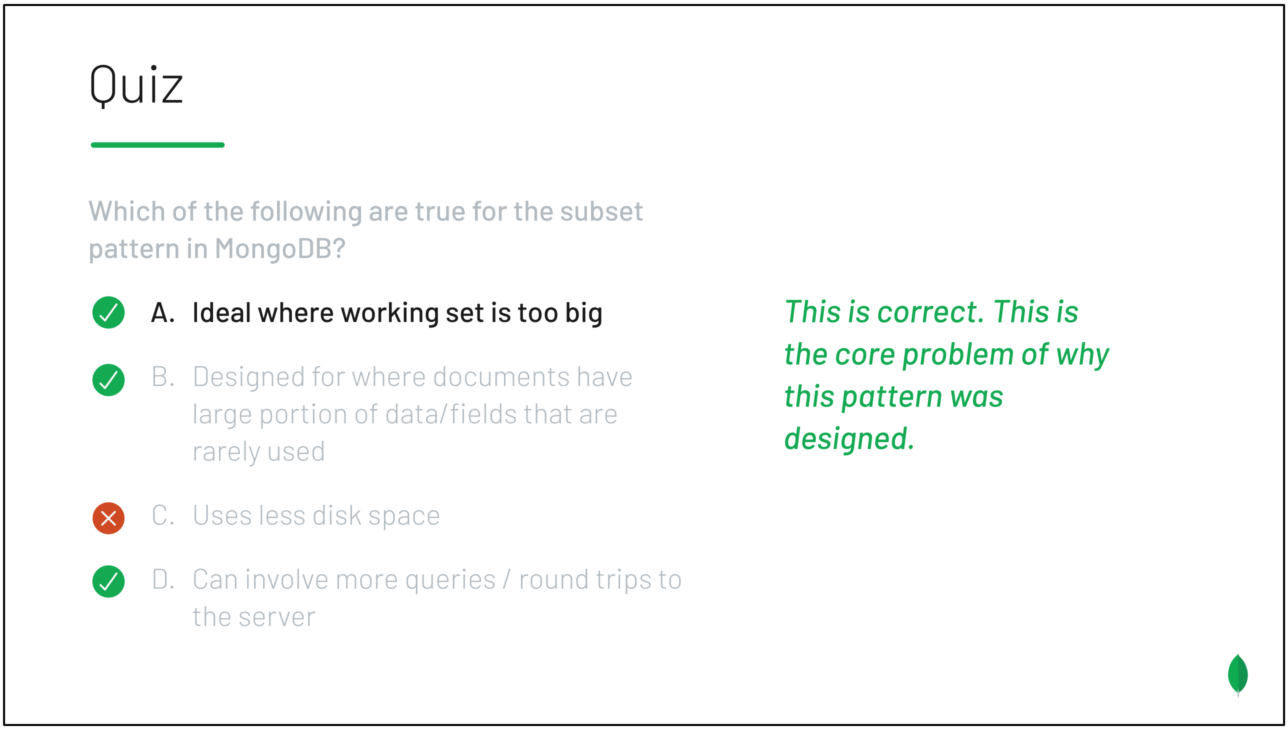
✓ A. Ideal where working set is too big

✓ B. Designed for where documents have large portion of data/fields that are rarely used

✗ C. Uses less disk space

✓ D. Can involve more queries / round trips to the server

CORRECT: Ideal where working set is too big. - This is the core problem of why this pattern was designed.
CORRECT: Designed for where documents have large portion of data/fields that are rarely used. - This correct and it’s the main symptom that the pattern tries to resolve.
INCORRECT: Uses less disk space. - This is incorrect splitting into two collections does increase the disk space.
CORRECT: Can involve more queries / round trips to the server. - If parts of the lesser accessed collection is required then more queries will be required.



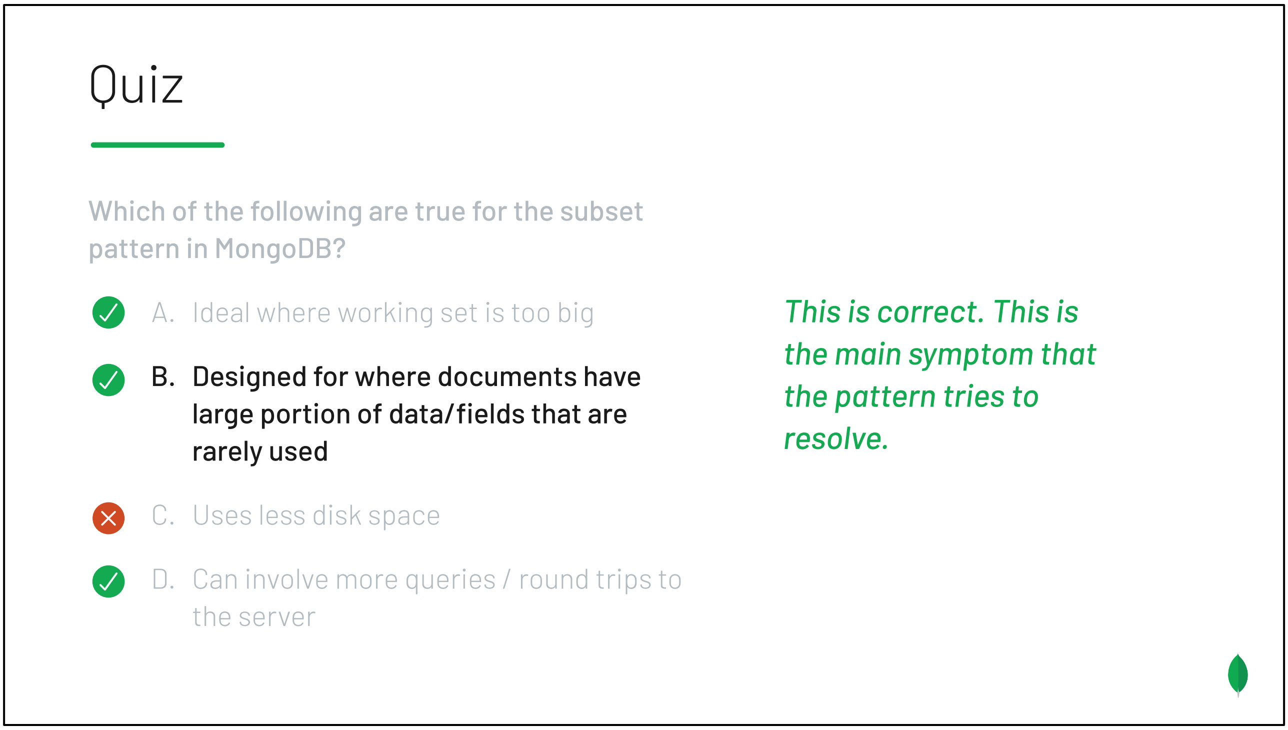
Quiz

Which of the following are true for the subset pattern in MongoDB?

- ✓ A. **Ideal where working set is too big**
- ✓ B. Designed for where documents have large portion of data/fields that are rarely used
- ✗ C. Uses less disk space
- ✓ D. Can involve more queries / round trips to the server

This is correct. This is the core problem of why this pattern was designed.

CORRECT: Ideal where working set is too big. - This is correct. This is the core problem of why this pattern was designed.



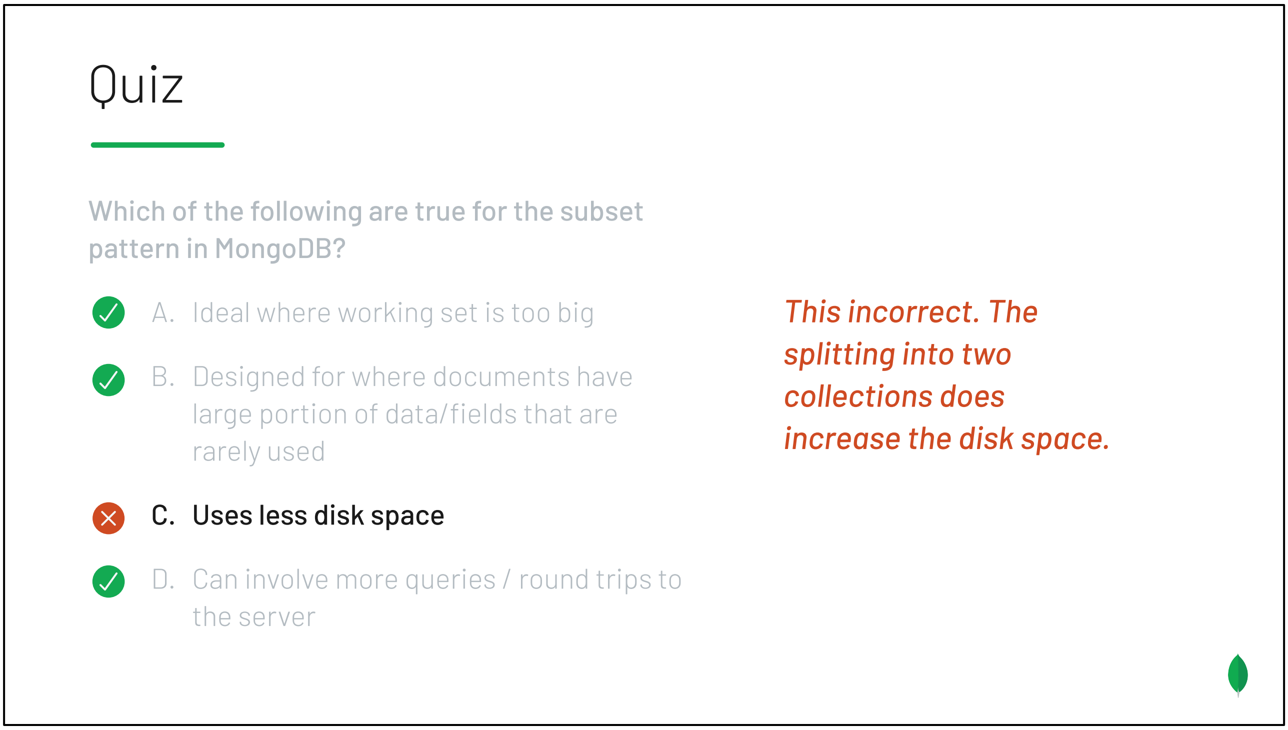
Quiz

Which of the following are true for the subset pattern in MongoDB?

- ✓ A. Ideal where working set is too big
- ✓ B. Designed for where documents have large portion of data/fields that are rarely used
- ✗ C. Uses less disk space
- ✓ D. Can involve more queries / round trips to the server

This is correct. This is the main symptom that the pattern tries to resolve.

CORRECT: Designed for where documents have large portion of data/fields that are rarely used. - This is correct. This is the main symptom that the pattern tries to resolve.



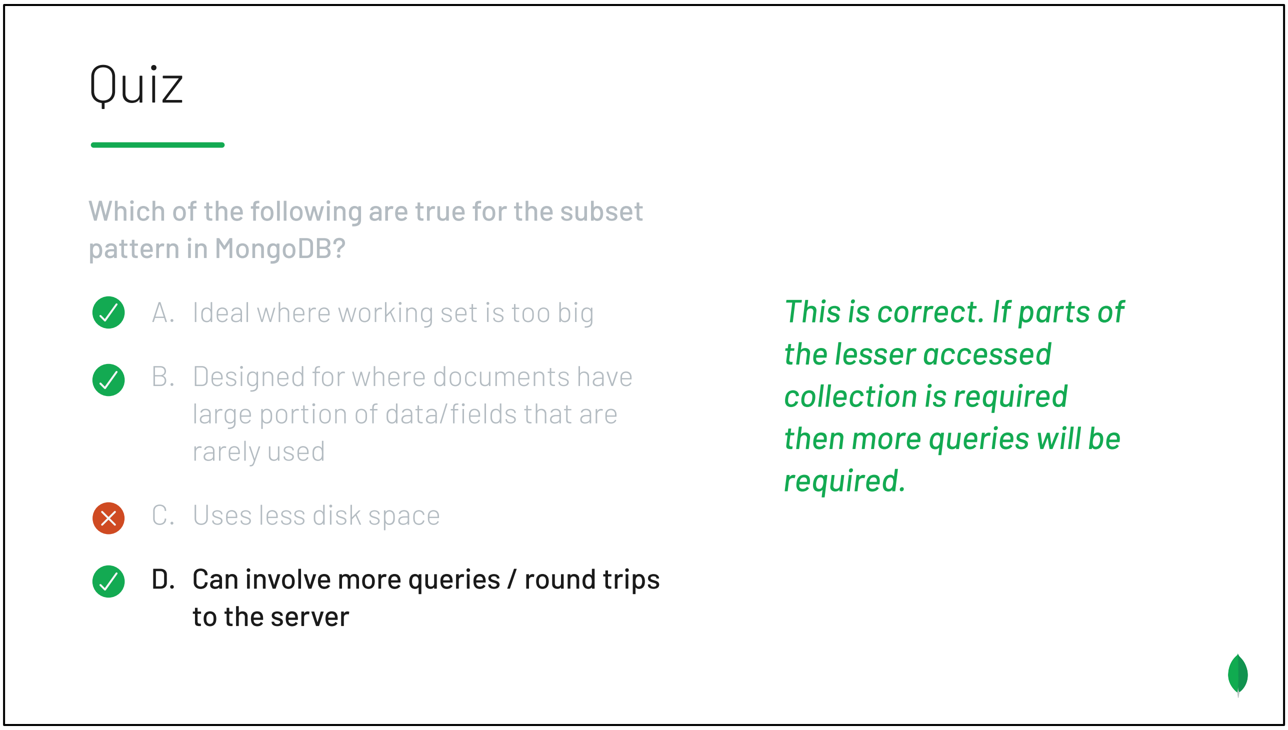
Quiz

Which of the following are true for the subset pattern in MongoDB?

- ✓ A. Ideal where working set is too big
- ✓ B. Designed for where documents have large portion of data/fields that are rarely used
- ✗ C. Uses less disk space
- ✓ D. Can involve more queries / round trips to the server

This incorrect. The splitting into two collections does increase the disk space.

INCORRECT: Uses less disk space. - This is incorrect. The splitting into two collections does increase the disk space.



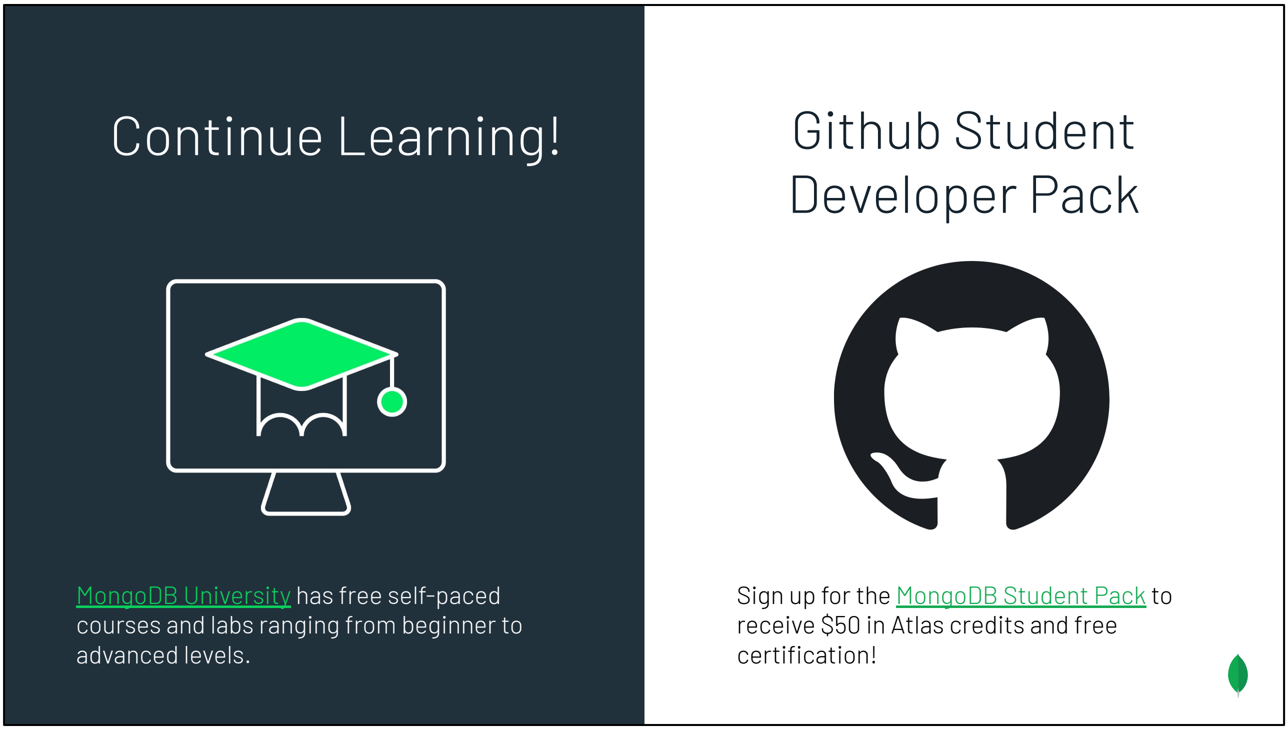
Quiz

Which of the following are true for the subset pattern in MongoDB?

- ✓ A. Ideal where working set is too big
- ✓ B. Designed for where documents have large portion of data/fields that are rarely used
- ✗ C. Uses less disk space
- ✓ **D. Can involve more queries / round trips to the server**

This is correct. If parts of the lesser accessed collection is required then more queries will be required.

CORRECT: Can involve more queries / round trips to the server. - This is correct. If parts of the lesser accessed collection is required then more queries will be required.



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.