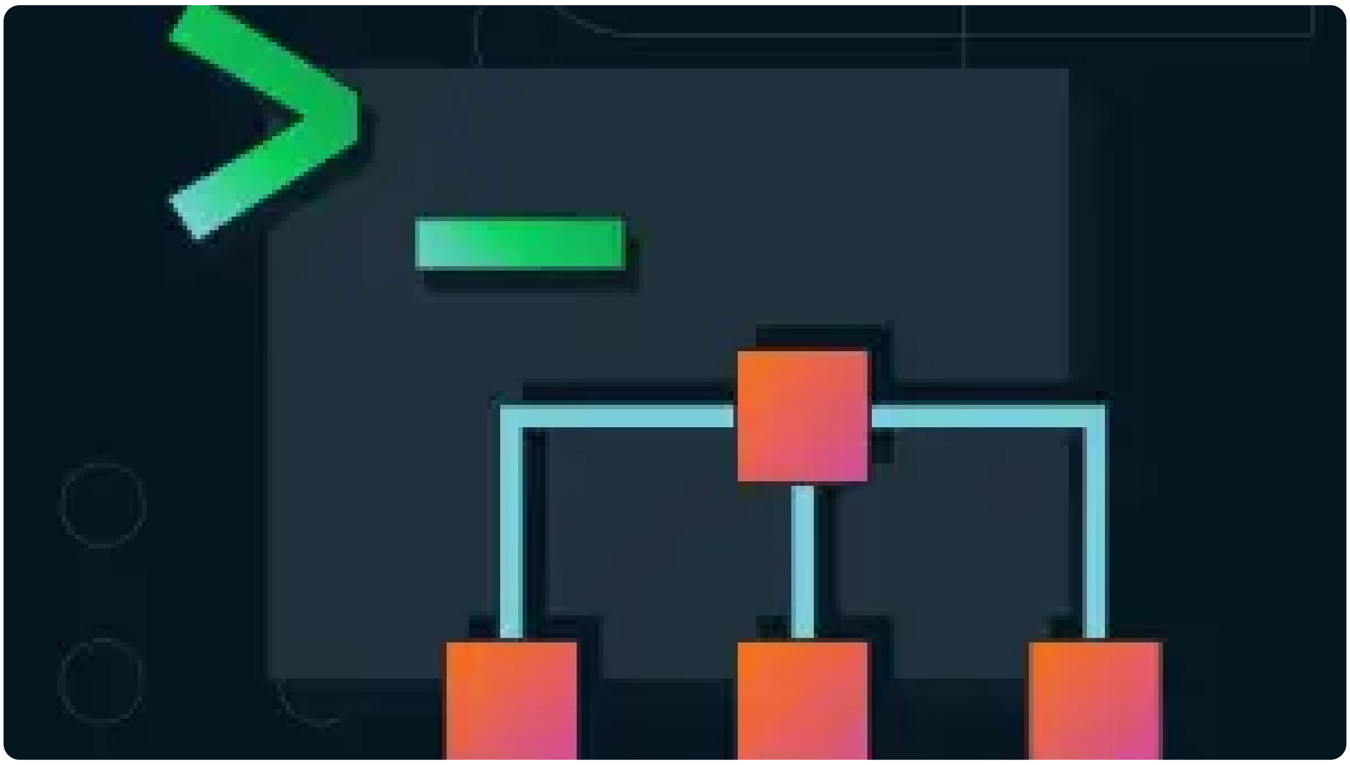
MongoDB

MongoDB Schema Design Best Practices

Joe Karlsson

11 min read • Published Jan 11, 2022 • Updated May 31, 2022

MongoDB Schema



Rate this tutorial ☆ ☆ ☆ ☆ ☆

Have you ever wondered, "How do I model a schema for my application?" It's one of the most common questions devs have pertaining to MongoDB. And the answer is, *it depends*. This is because document databases have a rich vocabulary that is capable of expressing data relationships in more nuanced ways than SQL. There are many things to consider when picking a schema. Is your app read or write heavy? What data is frequently accessed together? What are your performance considerations? How will your data set grow and scale?

In this post, we will discuss the basics of data modeling using real world examples. You will learn common methodologies and vocabulary you can use when designing your database schema for your application.

Okay, first off, did you know that proper MongoDB schema design is the most critical part of deploying a scalable, fast, and affordable database? It's true, and schema design is often one of the most overlooked facets of MongoDB administration. Why is MongoDB Schema Design so important? Well, there are a couple of good reasons. In my experience, most people coming to MongoDB tend to think of MongoDB schema design as the same as legacy relational schema design, which doesn't allow you to take full advantage of all that MongoDB databases have to offer. First, let's look at how legacy relational database design compares to MongoDB schema design.

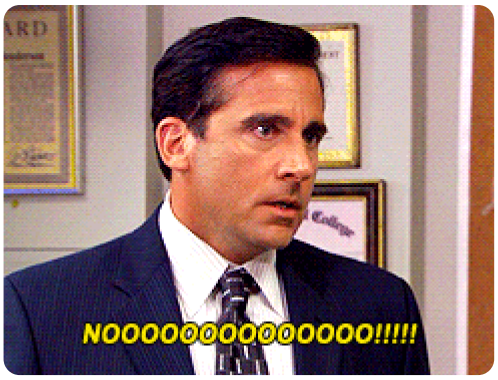
Schema Design Approaches – Relational vs. MongoDB

When it comes to MongoDB database schema design, this is what most developers think of when they are looking at designing a relational schema and a MongoDB schema.

RELATIONAL SCHEMA DESIGN
MONGODB SCHEMA DESIGN
Corporate needs you to find the differences between this picture and this picture.
They're the same picture.
imgflip.com

I have to admit that I understand the impulse to design your MongoDB schema the same way you have always designed your SQL schema. It's completely normal to want to split up your data into neat little tables as you've always done before. I was guilty of doing this when I first started learning how to use MongoDB. However, as we will soon see, you lose out on many of the awesome features of MongoDB when you design your schema like an SQL schema.

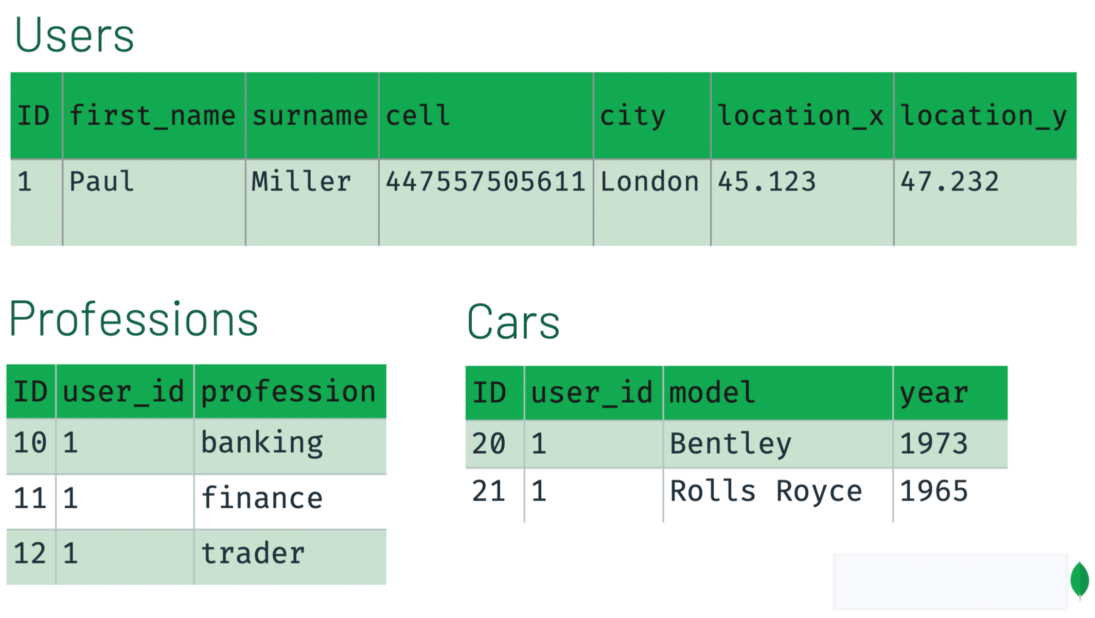
And this is how that makes me feel.



However, I think it's best to compare MongoDB schema design to relational schema design since that's where many devs coming to MongoDB are coming from. So, let's see how these two design patterns differ.

Relational Schema Design

When designing a relational schema, typically, devs model their schema independent of queries. They ask themselves, "What data do I have?" Then, by using prescribed approaches, they will normalize (typically in 3rd normal form). The tl;dr of normalization is to split up your data into tables, so you don't duplicate data. Let's take a look at an example of how you would model some user data in a relational database.



Users

ID	first_name	surname	cell	city	location_x	location_y
1	Paul	Miller	447557505611	London	45.123	47.232

Professions

ID	user_id	profession
10	1	banking
11	1	finance
12	1	trader

Cars

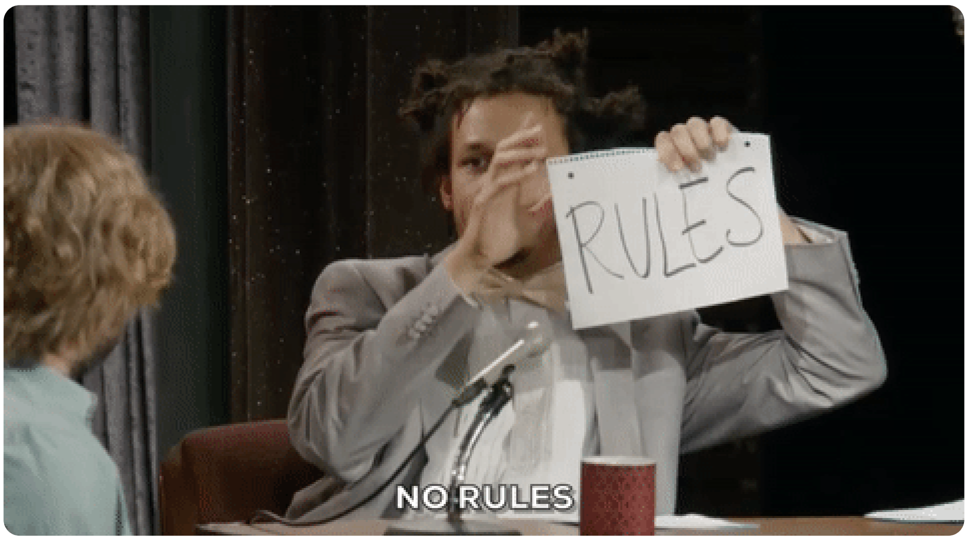
ID	user_id	model	year
20	1	Bentley	1973
21	1	Rolls Royce	1965

In this example, you can see that the user data is split into separate tables and it can be JOINED together using foreign keys in the `user_id` column of the Professions and Cars table. Now, let's take a look at how we might model this same data in MongoDB.

MongoDB Schema Design

Now, MongoDB schema design works a lot differently than relational schema design. With MongoDB schema design, there is:

- No formal process
- No algorithms
- No rules



When you are designing your MongoDB schema design, the only thing that matters is that you design a schema that will work well for ***your*** application. Two different apps that use the same exact data might have very different schemas if the applications are used differently. When designing a schema, we want to take into consideration the following:

- Store the data
- Provide good query performance

- Require reasonable amount of hardware

Let's take a look at how we might model the relational User model in MongoDB.

```
{
    "first_name": "Paul",
    "surname": "Miller",
    "cell": "447557505611",
    "city": "London",
    "location": [45.123, 47.232],
    "profession": ["banking", "finance", "trader"],
    "cars": [
        {
            "model": "Bentley",
            "year": 1973
        },
        {
            "model": "Rolls Royce",
            "year": 1965
        }
    ]
}
```

You can see that instead of splitting our data up into separate collections or documents, we take advantage of MongoDB's document based design to embed data into arrays and objects within the User object. Now we can make one simple query to pull all that data together for our application.

Embedding vs. Referencing

MongoDB schema design actually comes down to only two choices for every piece of data. You can either embed that data directly or reference another pros and cons of using each option in your schema.

Embedding

Advantages

- You can retrieve all relevant information in a single query.
- Avoid implementing joins in application code or using $lookup.
- Update related information as a single atomic operation. By default, all CRUD operations on a single document are ACID compliant.
- However, if you need a transaction across multiple operations, you can use the transaction operator.
- Though transactions are available starting 4.0, however, I should add that it's an anti-pattern to be overly reliant on using them in your application.

Limitations

- Large documents mean more overhead if most fields are not relevant. You can increase query performance by limiting the size of the documents that you are sending over the wire for each query.
- There is a 16-MB document size limit in MongoDB. If you are embedding too much data inside a single document, you could potentially hit this limit.

Referencing

Okay, so the other option for designing our schema is referencing another document using a document's unique object ID and connecting them together using the $lookup operator. Referencing works similarly as the JOIN operator in an SQL query. It allows us to split up data to make more efficient and scalable queries, yet maintain relationships between data.

Advantages

- By splitting up data, you will have smaller documents.
- Less likely to reach 16-MB-per-document limit.
- Infrequently accessed information not needed on every query.
- Reduce the amount of duplication of data. However, it's important to note that data duplication should not be avoided if it results in a better schema.

Limitations

- In order to retrieve all the data in the referenced documents, a minimum of two queries or $lookup required to retrieve all the information.

Type of Relationships

Okay, so now that we have explored the two ways we are able to split up data when designing our schemas in MongoDB, let's look at common relationships that you're probably familiar with modeling if you come from an SQL background. We will start with the more simple relationships and work our way up to some interesting patterns and relationships and how we model them with real-world examples. Note, we are only going to scratch the surface of modeling relationships in MongoDB here.

It's also important to note that even if your application has the same exact data as the examples listed below, you might have a completely different schema than the one I outlined here. This is because the most important consideration you make for your schema is how your data is going to be used by your system. In each example, I will outline the requirements for each application and why a given schema was used for that example. If you want to discuss the specifics of your schema, be sure to open a conversation on the MongoDB Community Forum, and we all can discuss what will work best for your unique application.

One-to-One

Let's take a look at our User document. This example has some great one-to-one data in it. For example, in our system, one user can only have one name. So, this would be an example of a one-to-one relationship. We can model all one-to-one data as key-value pairs in our database.

```
{
    "_id": "ObjectId('AAA')",
    "name": "Joe Karlsson",
    "company": "MongoDB",
    "twitter": "@JoeKarlsson1",
    "twitch": "joe_karlsson",
    "tiktok": "joekarlsson",
    "website": "joekarlsson.com"
}
```

DJ Khalid would approve.

One to One tl;dr:

- Prefer key-value pair embedded in the document.
- For example, an employee can work in one and only one department.

One-to-Few

Okay, now let's say that we are dealing a small sequence of data that's associated with our users. For example, we might need to store several addresses associated with a given user. It's unlikely that a user for our application would have more than a couple of different addresses. For relationships like this, we would define this as a *one-to-few relationship.*

```
{
    "_id": "ObjectId('AAA')",
    "name": "Joe Karlsson",
    "company": "MongoDB",
    "twitter": "@JoeKarlsson1",
    "twitch": "joe_karlsson",
    "tiktok": "joekarlsson",
    "website": "joekarlsson.com",
    "addresses": [
        { "street": "123 Sesame St", "city": "Anytown", "cc": '
        { "street": "123 Avenue Q",  "city": "New York", "cc":
    ]
}
```

Remember when I told you there are no rules to MongoDB schema design? Well, I lied. I've made up a couple of handy rules to help you design your schema for your application.

> **Rule 1**: Favor embedding unless there is a compelling reason not to.

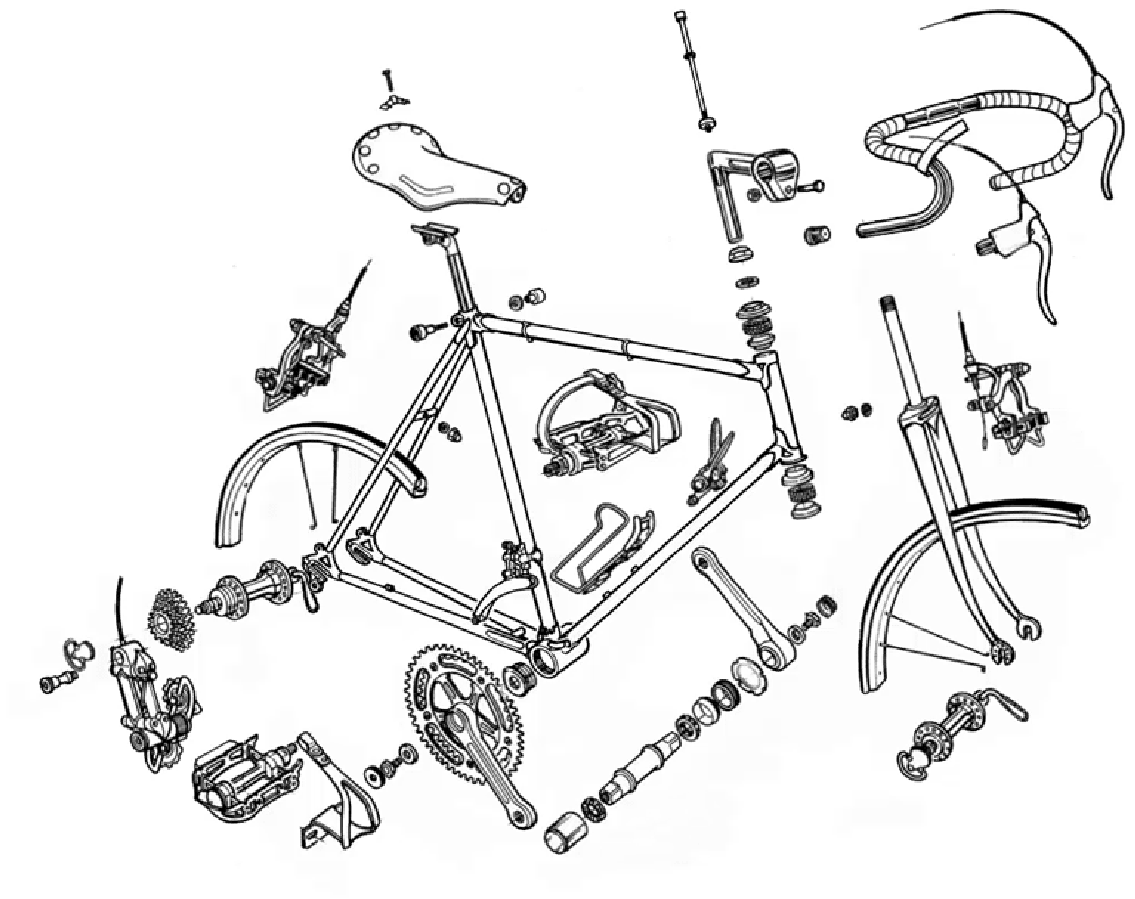
Generally speaking, my default action is to embed data within a document. I pull it out and reference it only if I need to access it on its own, it's too big, I rarely need it, or any other reason.

One-to-few tl;dr:

- Prefer embedding for one-to-few relationships.

One-to-Many

Alright, let's say that you are building a product page for an e-commerce website, and you are going to have to design a schema that will be able to show product information. In our system, we save information about all the many parts that make up each product for repair services. How would you design a schema to save all this data, but still make your product page performant? You might want to consider a *one-to-many* schema since your one product is made up of many parts.



Now, with a schema that could potentially be saving thousands of sub parts, we probably do not need to have all of the data for the parts on every single request, but it's still important that this relationship is maintained in our schema. So, we might have a Products collection with data about each product in our e-commerce store, and in order to keep that part data linked, we can keep an array of Object IDs that link to a document that has information about the part. These parts can be saved in the same collection or in a separate collection, if needed. Let's take a look at how this would look.

Products:

```
{
    "name": "left-handed smoke shifter",
    "manufacturer": "Acme Corp",
    "catalog_number": "1234",
    "parts": ["ObjectID('AAAA')", "ObjectID('BBBB')", "ObjectII
}
```

Parts:

```
{
    "_id" : "ObjectID('AAAA')",
    "partno" : "123-aff-456",
    "name" : "#4 grommet",
    "qty": "94",
    "cost": "0.94",
    "price":" 3.99"
}
```

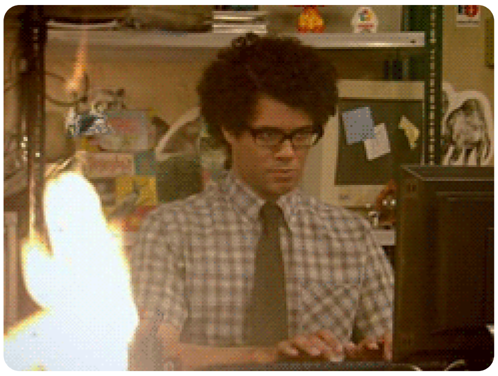
Rule 2: Needing to access an object on its own is a compelling reason not to embed it.

Rule 3: Avoid joins/lookups if possible, but don't be afraid if they can provide a better schema design.

One-to-Squillions

What if we have a schema where there could be potentially millions of subdocuments, or more? That's when we get to the one-to-squillions schema. And, I know what you're thinking: _Is squillions a real word?_
And the answer is yes, it is a real word.

Let's imagine that you have been asked to create a server logging application. Each server could potentially save a massive amount of data, depending on how verbose you're logging and how long you store server logs for.



With MongoDB, tracking data within an unbounded array is dangerous, since we could potentially hit that 16-MB-per-document limit. Any given host could generate enough messages to overflow the 16-MB document size, even if only ObjectIDs are stored in an array. So, we need to rethink how we can track this relationship without coming up against any hard limits.

So, instead of tracking the relationship between the host and the log message in the host document, let's let each log message store the host that its message is associated with. By storing the data in the log, we no longer need to worry about an unbounded array messing with our application! Let's take a look at how this might work.

Hosts:

```
{
    "_id": ObjectID("AAAB"),
    "name": "goofy.example.com",
    "ipaddr": "127.66.66.66"
}
```

Log Message:

```
{
    "time": ISODate("2014-03-28T09:42:41.382Z"),
    "message": "cpu is on fire!",
    "host": ObjectID("AAAB")
}
```

Rule 4: Arrays should not grow without bound. If there are more than a couple of hundred documents on the "many" side, don't embed them; if there are more than a few thousand documents on the "many" side, don't use an array of ObjectID references. High-cardinality arrays are a compelling reason not to embed.

Many-to-Many

The last schema design pattern we are going to be covering in this post is the *many-to-many* relationship. This is another very common schema pattern that we see all the time in relational and MongoDB schema designs. For this pattern, let's imagine that we are building a to-do application. In our app, a user may have *many* tasks and a task may have *many* users assigned to it.

In order to preserve these relationships between users and tasks, there will need to be references from the *one* user to the *many* tasks and references from the *one* task to the *many* users. Let's look at how this could work for a to-do list application.

Users:

```
{
    "_id": ObjectID("AAF1"),
    "name": "Kate Monster",
    "tasks": [ObjectID("ADF9"), ObjectID("AE02"), ObjectID("AE7
}
```

Tasks:

```
{
    "_id": ObjectID("ADF9"),
    "description": "Write blog post about MongoDB schema design
    "due_date": ISODate("2014-04-01"),
    "owners": [ObjectID("AAF1"), ObjectID("BB3G")]
}
```

From this example, you can see that each user has a sub-array of linked tasks, and each task has a sub-array of owners for each item in our to-do app.

Summary

As you can see, there are a ton of different ways to express your schema design, by going beyond normalizing your data like you might be used to doing in SQL. By taking advantage of embedding data within a document or referencing documents using the $lookup operator, you can make some truly powerful, scalable, and efficient database queries that are completely unique to your application. In fact, we are only barely able to scratch the surface of all the ways that you could model your data in MongoDB. If you want to learn more about MongoDB schema design, be sure to check out our continued series on schema design in MongoDB:

- MongoDB schema design anti-patterns
- MongoDB University - M320: Data Modeling
- MongoDB Data Model Design Documentation
- Building with Patterns: A Summary

I want to wrap up this post with the most important rule to MongoDB schema design yet.

> **Rule 5**: As always, with MongoDB, how you model your data depends – entirely – on your particular application's data access patterns. You want to structure your data to match the ways that your application queries and updates it.

Remember, every application has unique needs and requirements, so the schema design should reflect the needs of that particular application. Take the examples listed in this post as a starting point for your application. Reflect on what you need to do, and how you can use your schema to help you get there.

> Recap:
>
> - **One-to-One** - Prefer key value pairs within the document
> - **One-to-Few** - Prefer embedding
> - **One-to-Many** - Prefer embedding
> - **One-to-Squillions** - Prefer Referencing
> - **Many-to-Many** - Prefer Referencing

> General Rules for MongoDB Schema Design:
>
> - **Rule 1:** Favor embedding unless there is a compelling reason not to.

- **Rule 2**: Needing to access an object on its own is a compelling reason not to embed it.
- **Rule 3**: Avoid joins and lookups if possible, but don't be afraid if they can provide a better schema design.
- **Rule 4**: Arrays should not grow without bound. If there are more than a couple of hundred documents on the *many* side, don't embed them; if there are more than a few thousand documents on the *many* side, don't use an array of ObjectID references. High-cardinality arrays are a compelling reason not to embed.
- **Rule 5**: As always, with MongoDB, how you model your data depends **entirely** on your particular application's data access patterns. You want to structure your data to match the ways that your application queries and updates it.

We have only scratched the surface of design patterns in MongoDB. In fact, we haven't even begun to start exploring patterns that aren't even remotely possible to perform in a legacy relational model. If you want to learn more about these patterns, check out the resources below.

Additional Resources:

- Now that you know how to design a scalable and performant MongoDB schema, check out our MongoDB schema design anti-pattern series to learn what NOT to do when building out your MongoDB database schema: https://developer.mongodb.com/article/schema-design-anti-pattern-massive-arrays
- Video more your thing? Check out our video series on YouTube to learn more about MongoDB schema anti-patterns: https://www.youtube.com/watch?v=8CZs-0it9r4&feature=youtu.be
- MongoDB University - M320: Data Modeling
- 6 Rules of Thumb for MongoDB Schema Design: Part 1
- MongoDB Data Model Design Documentation

- MongoDB Data Model Examples and Patterns Documentation
- Building with Patterns: A Summary

Rate this tutorial ☆ ☆ ☆ ☆ ☆

Related

ARTICLE

Structuring Data with Serde in Rust

May 12, 2022 | 5 min read

ARTICLE

MongoDB Performance Tuning Questions

Sep 23, 2022 | 10 min read

TUTORIAL

Client-Side Field Level Encryption (CSFLE) in MongoDB with Golang

Feb 03, 2023 | 15 min read

ARTICLE

Window Functions & Time Series Collections

May 13, 2022 | 7 min read

Request a Tutorial

MongoDB.

About

Careers

Investor Relations

Legal Notices

Privacy Notices

Security Information

Trust Center

Support

Contact Us

Customer Portal

Atlas Status

Customer Support

Social

GitHub

Stack Overflow

LinkedIn

YouTube

Twitter

Twitch

Facebook