Lecture 8 - How to structure documents (schemas, and relations)

Instructor: Suresh Melvin Sigera
Email: suresh.sigera@cuny.csi.edu

Agenda

- Principles of schema design
- What are your application access patterns
- Understanding relations
    - One to one
    - One to many
    - Many to many
- QA

Schemas - Principles of schema design

Database schema design is the process of **choosing the best representation for a data set**, given the features of the database system, the nature of the data, and the application requirements.

To get you thinking, here are a few questions you can bring to the table when modeling data with any database system:

- What are your application access patterns?
- What’s the basic unit of data?
- What are the capabilities of your database?
- What makes a good unique id or primary key for a record?

Schemas - What are your application access patterns

You need to pin down the needs of your application, and this should inform not only your schema design but also which database you choose. **Remember, MongoDB isn’t right for every application**. Understanding your application access patterns is by far the most important aspect of schema design.

The idiosyncrasies of an application can easily demand a schema that goes against firmly held data modeling principles. The upshot is that you must ask numerous questions about the application before you can determine the ideal data model. What’s the **read/write ratio**? **Will queries be simple**, such as looking up a key, or more complex? Will aggregations be necessary? How much data will be stored?

Schemas - What’s the basic unit of data?

In an RDBMS, you have tables with columns and rows. In a key-value store, you have keys pointing to amorphous values. In MongoDB, the basic unit of data is the BSON document.

Schemas - What are the capabilities of your database?

Once you understand the basic data type, you need to know how to manipulate it. RDBMSs feature ad hoc queries and joins, usually written in SQL while simple key-value stores permit fetching values only by a single key. MongoDB also allows ad hoc queries, but joins aren't supported. Databases also diverge in the kinds of updates they permit. With an RDBMS, you can update records in sophisticated ways using SQL and wrap multiple updates in a transaction to get atomicity and rollback. MongoDB doesn't support transactions in the traditional sense, but it does support a variety of atomic update operations that can work on the internal structures of a complex document. With simple key-value stores, you might be able to update a value, but every update will usually mean replacing the value completely.

Schemas - What makes a good unique id or primary key for a record?

There are exceptions, but many schemas, regardless of the database system, have some unique key for each record. Choosing this key carefully can make a big difference in how you access your data and how it’s stored. If you’re designing a user’s collection, for example, should you use an arbitrary value, a legal name, a username, or a social security number as the primary key? It turns out that neither legal names nor social security numbers are unique or even applicable to all users within a given dataset. In MongoDB choosing a primary key means picking what should go in the _id field. The automatic object ids are good defaults, but not ideal in every case. This is particularly important if you shard your data across multiple machines because it determines where a certain record will go.

Schemas - What makes a good unique id or primary key for a record?

There are exceptions, but many schemas, regardless of the database system, have some unique key for each record. Choosing this key carefully can make a big difference in how you access your data and how it's stored. If you're designing a user's collection, for example, should you use an arbitrary value, a legal name, a username, or a social security number as the primary key? It turns out that neither legal names nor social security numbers are unique or even applicable to all users within a given dataset. In MongoDB choosing a primary key means picking what should go in the _id field. The automatic object ids are good defaults, but not ideal in every case. This is particularly important if you shard your data across multiple machines because it determines where a certain record will go.

Schemas - Note

The best schema designs are always the product of deep knowledge of the database you’re using, good judgment about the requirements of the application at hand, and plain old experience. **A good schema often requires experimentation and iteration**, such as when an application scales and performance considerations change.

Don’t be afraid to alter your schema when you learn new things; only rarely is it possible to fully plan an application before its implementation.

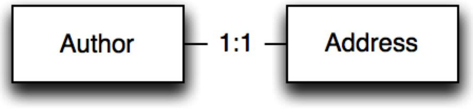
```
use shop

db.createCollection('products')

db.products.insertOne({ 'name' : 'Best of Python', 'price' : 49.99 })
db.products.insertOne({
     name: "Gildan Men's Assorted Crew T-Shirt Multipack",
     price: 11.99,
     seller:
         {
              name:"Gildan",
              category: "t-shirts",
         }
     })
```

Schemas - One-To-One (1:1)

The 1:1 relationship describes a relationship between two entities. In this case the Author has a single Address relationship where an Author lives at a single Address and an Address only contains a single Author.



Schemas - One-To-One (1:1)

The 1:1 relationship can be modeled in two ways using MongoDB. The first is to embed the relationship as a document, the second is as a link to a document in a separate collection. Let's look at both ways of modeling the one to one relationship using the following two documents:

```
{
  name: "Peter Wilkinson",
  age: 27
}
```

User document

```
{
  street: "100 some road",
  city: "Nevermore"
}
```

Address document

Schemas - Embedding

The first approach is simply to embed the **Address** document as **an embedded document in the User document**.

```
{
  name: "Peter Wilkinson",
  age: 27,
  address: {
    street: "100 some road",
    city: "Nevermore"
  }
}
```

Schemas - Embedding

The strength of **embedding** the **Address document directly in the User document** is that we can **retrieve the user and its addresses in a single read operation** versus having to first read the user document and then the address documents for that specific user. Since addresses have a strong affinity to the user document the embedding makes sense here.

Schemas - Linking

The second approach is to link the address and user document using a foreign key. This is similar to how traditional relational databases would store the data. It is important to note that MongoDB does not enforce any foreign key constraints so the relation only exists as part of the application level schema.

```
{
  _id: 1,
  name: "Peter Wilkinson",
  age: 27
}
```

User document

```
{
  user_id: 1,
  street: "100 some road",
  city: "Nevermore"
}
```

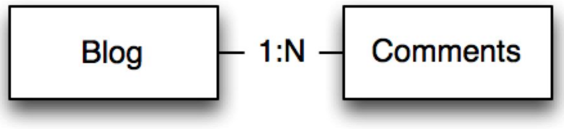
Address document

Schemas - Note

In the one to one relationship Embedding is the preferred way to model the relationship as it’s more efficient to retrieve the document.

Schemas - One-To-Many (1:N)

The 1:N relationship describes a relationship where one side can have more than one relationship while the reverse relationship can only be single sided. An example is a Blog where a blog might have many Comments but a Comment is only related to a single Blog.



Schemas - One-To-Many (1:N)

The 1:N relationship can be modeled in several different ways using MongoDB. In this chapter we will explore three different ways of modeling the 1:N relationship. The first is embedding, the second is linking and the third is a bucketing strategy that is useful for cases like time series. Let’s use the model of a **Blog Post** and its **Comments**.

Schemas - One-To-Many (1:N)

```
{
  title: "An awesome blog",
  url: "http://awesomeblog.com",
  text: "This is an awesome blog"
}
```

Blog document

```
{
  name: "Peter Critic",
  created_on:
ISODate("2014-01-01T10:01:22Z"),
  comment: "Awesome blog post"
}
{
  name: "John Page",
  created_on:
ISODate("2014-01-01T11:01:22Z"),
  comment: "Not so awesome blog"
}
```

Comment document

Schemas - Embedding

The first approach is to embed the Comments in the Blog post.

Schemas - Embedding

```
{
  title: "An awesome blog",
  url: "http://awesomeblog.com",
  text: "This is an awesome blog we have just started",
  comments: [{
    name: "Peter Critic",
    created_on: ISODate("2014-01-01T10:01:22Z"),
    comment: "Awesome blog post"
  }, {
    name: "John Page",
    created_on: ISODate("2014-01-01T11:01:22Z"),
    comment: "Not so awesome blog"
  }]
}
```

Schemas - Embedding

The embedding of the comments in the Blog post means we can easily retrieve all the comments belong to a particular Blog post. Adding new comments is as simple as appending the new comment document to the end of the comments array.

Schemas - Embedding

However, there are three potential problems associated with this approach that one should be aware off.

- The first is that the comments array might grow larger than the maximum document size of 16 MB.
- The second aspects relates to write performance. As comments get added to Blog Post over time, it becomes hard for MongoDB to predict the correct document padding to apply when a new document is created. MongoDB would need to allocate new space for the growing document. In addition, it would have to copy the document to the new memory location and update all indexes. This could cause a lot more IO load and could impact overall write performance.

Schemas - Linking

The second approach is to link comments to the Blog Post using a more traditional foreign key.

```
{
  _id: 1,
  title: "An awesome blog",
  url: "http://awesomeblog.com",
  text: "This is an awesome blog"
}
```

Blog post document

```
{
  blog_entry_id: 1,
  name: "Peter Critic",
  created_on:
ISODate("2014-01-01T10:01:22Z"),
  comment: "Awesome blog post"
}
{
  blog_entry_id: 1,
  name: "John Page",
  created_on:
ISODate("2014-01-01T11:01:22Z"),
  comment: "Not so awesome blog"
}
```

Comment document

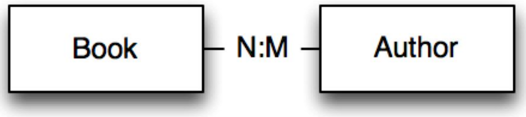
Schemas - Linking

An advantage this model has is that additional comments will not grow the original Blog Post document, making it less likely that the applications will run in the maximum document size of 16 MB.

It’s also much easier to return paginated comments as the application can slice and dice the comments more easily. On the downside if we have 1000 comments on a blog post, we would need to retrieve all 1000 documents causing a lot of reads from the database.

Schemas - Many-To-Many (N:M)

An N:M relationship is an example of a relationship between two entities where they both might have many relationships between each other. An example might be a Book that was written by many Authors. At the same time an Author might have written many Books.



Schemas - Many-To-Many (N:M)

N:M relationships are modeled in the relational database by using a join table. A good example is the relationship between books and authors.

- An author might have authored multiple books (1:N).
- A book might have multiple authors (1:M).

This leads to an N:M relationship between authors of books.

Schemas - Two Way Embedding

Embedding the books in an Author document. In Two Way Embedding we will include the Book foreign keys under the books field. Mirroring the Author document, for each Book we include the Author foreign keys under the Author field.

```
{
  _id: 1,
  name: "Peter Standford",
  books: [1, 2]
}
{
  _id: 2,
  name: "Georg Peterson",
  books: [2]
}
```

Author document

```
{
  _id: 1,
  title: "A tale of two people",
  categories: ["drama"],
  authors: [1, 2]
}
{
  _id: 2,
  title: "A tale of two space
ships",
  categories: ["scifi"],
  authors: [1]
}
```

Comment document

QA