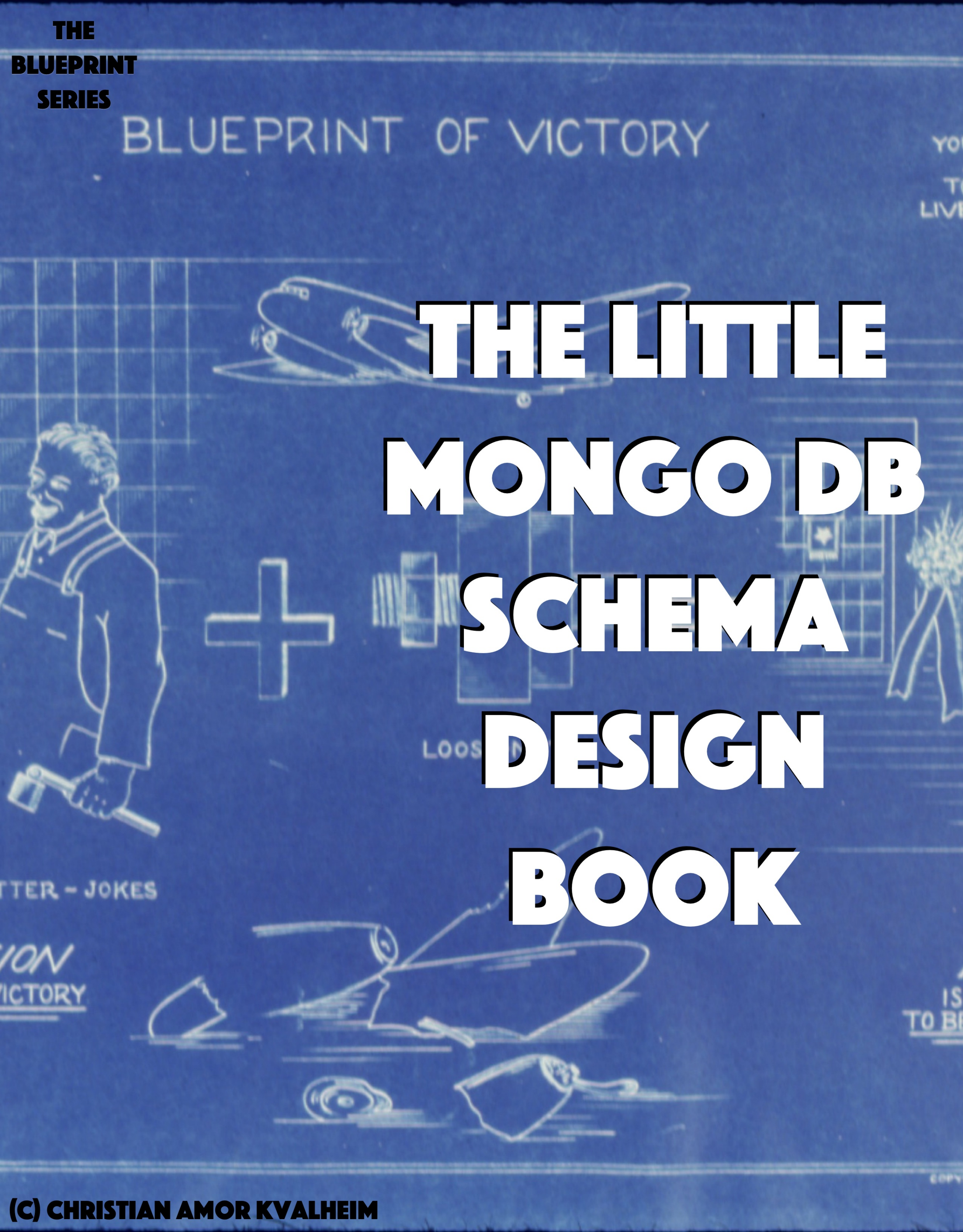
THE BLUEPRINT SERIES



THE LITTLE MONGO DB SCHEMA DESIGN BOOK

The Little Mongo DB Schema Design Book

Christian Kvalheim

This book is for sale at http://leanpub.com/mongodbschemadesign

This version was published on 2015-10-09

ISBN 978-1-943846-77-1

Tweet This Book!

Please help Christian Kvalheim by spreading the word about this book on Twitter!

The suggested hashtag for this book is #mongodbschema.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#mongodbschema

Contents

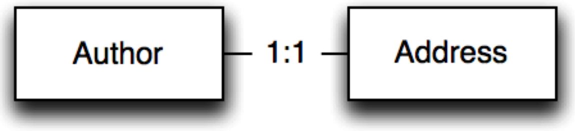
Schema Basics

Before exploring the more advanced schemas in this book it's important to revisit schema basics. In this chapter we will explore the basic relationships from traditional relational databases and how they relate to the document model in MongoDB.

We will start with a look at the One-To-One (1:1) relationship then moving on to the One-To-Many (1:N) and finally the Many-To-Many (N:M).

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*.



A One to One Relational Example

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:

Model

An example User document

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

An example Address document

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

Embedding

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

An example User document with Embedded Address

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

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.

Linking

The second approach is to link the address and user document using a `foreign key`.

An example User document

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

An example Address document with Foreign Key

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

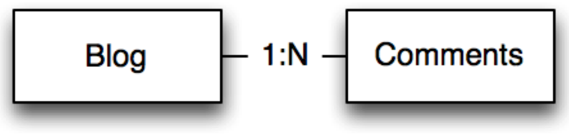
This is similar to how traditional relational databases would store the data. It is important to note that MongoDB does not enforce any foreign key constraints so the relation only exists as part of the application level schema.

`Embedding Preferred`

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

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*.



A One to Many Relational Example

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`.

Model

An example Blog Post document

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

A Blog Post is a single document that describes one specific blog post.

Some example Comment documents

```
{
  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"
}
```

For each Blog Post we can have one or more Comments.

Embedding

The first approach is to embed the Comments in the `Blog Post`.

A Blog Post with Embedded documents

```
{
  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"
  }]
}
```

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.

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.

It's important to note that this only matters for *high write* traffic and might not be a problem for smaller applications. What might not be acceptable for a *high write* volume application might be tolerable for an application with *low write* load.

- The third problem is exposed when one tries to perform pagination of the comments. There is no way to limit the comments returned from the `Blog Post` using normal finds so we will have to retrieve the whole blog document and filter the comments in our application.

Linking

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

An example Blog Post document

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

Some example Comment documents with Foreign Keys

```
{
  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"
}
```

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.

Bucketing

The third approach is a hybrid of the two above. Basically, it tries to balance the rigidity of the embedding strategy with the flexibility of the linking strategy. For this example, we will split the comments into buckets with a maximum of 50 comments in each bucket.

An example Blog Post document

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

Some example Comment buckets

```
{
  blog_entry_id: 1,
  page: 1,
  count: 50,
  comments: [{
    name: "Peter Critic",
    created_on: ISODate("2014-01-01T10:01:22Z"),
    comment: "Awesome blog post"
  }, ...]
}

{
  blog_entry_id: 1,
  page: 2,
  count: 1,
  comments: [{
    name: "John Page",
    created_on: ISODate("2014-01-01T11:01:22Z"),
    comment: "Not so awesome blog"
  }]
}
```

The main benefit of using buckets in this case is that we can perform a single read to fetch 50 comments at a time, allowing for efficient pagination.

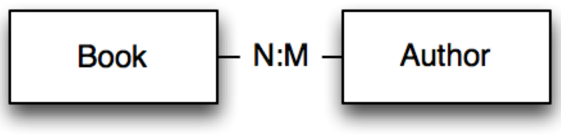
`When to use bucketing`

When you have the possibility of splitting up your documents into discreet batches, it makes sense to consider bucketing to speed up document retrieval.

Typical cases where bucketing is appropriate are ones such as bucketing data by hours, days or number of entries on a page (like comments pagination).

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*.



A Many to Many Relational Example

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. Let's look at how this can be modeled.

Two Way Embedding

Embedding the books in an Author document

Model

In `Two Way Embedding` we will include the *Book* `foreign keys` under the `books` field.

Some example Author documents

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

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

Mirroring the *Author* document, for each *Book* we include the *Author* `foreign keys` under the *Author* field.

Some example Book documents

```
{
  _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]
}
```

Queries

Example 1: Fetch books by a specific author

```
var db = db.getSisterDB("library");
var booksCollection = db.books;
var authorsCollection = db.authors;

var author = authorsCollection.findOne({name: "Peter Standford"});
var books = booksCollection.find({_id: {$in: author.books}}).toArray\
();
```

Example 2: Fetch authors by a specific book

```
var db = db.getSisterDB("library");
var booksCollection = db.books;
var authorsCollection = db.authors;

var book = booksCollection.findOne({title: "A tale of two space ship\
s"});
var authors = authorsCollection.find({_id: {$in: book.authors}}).toA\
rray();
```

As can be seen, we have to perform two queries in both directions. The first is to find either the author or the book and the second is to perform a $in query to find the books or authors.

`Uneven n:m relationships`

Letâ€™s take the category `drama` that might have thousands of books in it and with many new books consistently being added and removed. This makes it impracticable to embed all the books in a category document. Each book, however, can easily have categories embedded within it, as the rate of change of categories for a specific book might be very low.

In this case we should consider `One way embedding` as a strategy.

One Way Embedding

The One Way Embedding strategy chooses to optimize the read performance of a *N:M* relationship by embedding the references in one side of the relationship. An example might be where several books belong to a few categories but a couple categories have many books. Letâ€™s look at an example, pulling the categories out into a separate document.

Model

A Category document

```
{
  _id: 1,
  name: "drama"
}
```

A Book with Foreign Keys for Categories

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

The reason we choose to embed all the references to categories in the books is due to there being lot more books in the drama category than categories in a book. If one embeds the books in the category document it's easy to foresee that one could break the 16MB max document size for certain broad categories.

Queries

Example 3: Fetch categories for a specific book

```
var db = db.getSisterDB("library");
var booksCol = db.books;
var categoriesCol = db.categories;

var book = booksCol.findOne({title: "A tale of two space ships"});
var categories = categoriesCol.find({_id: {$in: book.categories}}).t\
oArray();
```

Example 4: Fetch books for a specific category

```
var db = db.getSisterDB("library");
var booksCollection = db.books;
var categoriesCollection = db.categories;

var category = categoriesCollection.findOne({name: "drama"});
var books = booksCollection.find({categories: category.id}).toArray(\
);
```

`Establish Relationship Balance`

Establish the maximum size of `N` and the size of `M`. For example if `N` is a maximum of 3 categories for a book and `M` is a maximum of 500000 books in a category you should pick One Way Embedding. If `N` is a maximum of 3 and `M` is a maximum of 5 then Two Way Embedding might work well.