6 Rules of Thumb for MongoDB Schema Design: Part 1

By William Zola, Lead Technical Support Engineer at MongoDB

MongoDB has a rich and nuanced vocabulary for expressing what, in SQL, gets flattened into the term “One-to-N”. Let me take you on a tour of your choices in modeling One-to-N relationships.

There’s so much to talk about here, I’m breaking this up into three parts. In this first part, I’ll talk about the three basic ways to model One-to-N relationships. In the second part I’ll cover more sophisticated schema designs, including denormalization and two-way referencing. And in the final part, I’ll review the entire rainbow of choices, and give you some suggestions for choosing among the thousands (really — thousands) of choices that you may consider when modeling a single One-to-N relationship.

Many beginners think that the only way to model “One-to-N” in MongoDB is to embed an array of sub-documents into the parent document, but that’s just not true. Just because you can embed a document, doesn’t mean you should embed a document.

When designing a MongoDB schema, you need to start with a question that you’d never consider when using SQL: what is the cardinality of the relationship? Put less formally: you need to characterize your “One-to-N” relationship with a bit more nuance: is it “one-to-few”, “one-to-many”, or “one-to-squillions”? Depending on which one it is, you’d use a different format to model the relationship.

Basics: Modeling One-to-Few

An example of “one-to-few” might be the addresses for a person. This is a good use case for embedding — you’d put the addresses in an array inside of your Person object:

```
> db.person.findOne()

{

  name: 'Kate Monster',
  ssn: '123-456-7890',

  addresses : [

     { street: '123 Sesame St', city: 'Anytown', cc: 'USA' },

     { street: '123 Avenue Q', city: 'New York', cc: 'USA' }

  ]

}
```

This design has all of the advantages and disadvantages of embedding. The main advantage is that you don't have to perform a separate query to get the embedded details; the main disadvantage is that you have no way of accessing the embedded details as stand-alone entities.

For example, if you were modeling a task-tracking system, each Person would have a number of Tasks assigned to them. Embedding Tasks inside the Person document would make queries like "Show me all Tasks due tomorrow" much more difficult than they need to be. I will cover a more appropriate design for this use case in the next post.

Basics: One-to-Many

An example of "one-to-many" might be parts for a product in a replacement parts ordering system. Each product may have up to several hundred replacement parts, but never more than a couple thousand or so. (All of those different-sized bolts, washers, and gaskets add up.) This is a good use case for referencing — you'd put the ObjectIDs of the parts in an array in product document. (For these examples I'm using 2-byte ObjectIDs because they're easier to read: real-world code would use 12-byte ObjectIDs.)

Each Part would have its own document:

```
> db.parts.findOne()

{

    _id : ObjectID('AAAA'),
    partno : '123-aff-456',

    name : '#4 grommet',

    qty: 94,

    cost: 0.94,

    price: 3.99

}
```

Each Product would have its own document, which would contain an array of ObjectID references to the Parts that make up that Product:

```
> db.products.findOne()

{

    name : 'left-handed smoke shifter',

    manufacturer : 'Acme Corp',

    catalog_number: 1234,

    parts : [     // array of references to Part documents

        ObjectID('AAAA'),    // reference to the #4 grommet above

        ObjectID('F17C'),    // reference to a different Part

        ObjectID('D2AA'),

        // etc

    ]
```

You would then use an **application-level join** to retrieve the parts for a particular product:

```
// Fetch the Product document identified by this catalog number
```

```
> product = db.products.findOne({catalog_number: 1234});


// Fetch all the Parts that are linked to this Product

> product_parts = db.parts.find({_id: { $in : product.parts } } )
.toArray() ;
```

For efficient operation, you'd need to have an index on 'products.catalog_number'. Note that there will always be an index on 'parts._id', so that query will always be efficient.

This style of referencing has a complementary set of advantages and disadvantages to embedding. Each Part is a stand-alone document, so it's easy to search them and update them independently. One trade off for using this schema is having to perform a second query to get details about the Parts for a Product. (But hold that thought until we get to denormalizing in part 2.)

As an added bonus, this schema lets you have individual Parts used by multiple Products, so your One-to-N schema just became an N-to-N schema without any need for a join table!

Basics: One-to-Squillions

An example of "one-to-squillions" might be an event logging system that collects log messages for different machines. Any given host could generate enough messages to overflow the 16 MB document size, even if all you stored in the array was the ObjectID. This is the classic use case for "parent-referencing" — you'd have a document for the host, and then store the ObjectID of the host in the documents for the log messages.

```
> db.hosts.findOne()

{

    _id : ObjectID('AAAB'),

    name : 'goofy.example.com',
    ipaddr : '127.66.66.66'
}
>db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    host: ObjectID('AAAB')      // Reference to the Host document
}
```

You’d use a (slightly different) application-level join to find the most recent 5,000 messages for a host:

```
  // find the parent ‘host’ document
> host = db.hosts.findOne({ipaddr : '127.66.66.66'});  // assumes
unique index
// find the most recent 5000 log message documents linked to that
host
> last_5k_msg = db.logmsg.find({host: host._id}).sort({time : -1}
).limit(5000).toArray()
```

Recap

So, even at this basic level, there is more to think about when designing a MongoDB schema than when designing a comparable relational schema. You need to consider two factors:

- Will the entities on the “N” side of the One-to-N ever need to stand alone?
- What is the cardinality of the relationship: is it one-to-few; one-to-many; or one-to-squillions?

Based on these factors, you can pick one of the three basic One-to-N schema designs:

- Embed the N side if the cardinality is one-to-few and there is no need to access the embedded object outside the context of the parent object
- Use an array of references to the N-side objects if the cardinality is one-to-many or if the N-side objects should stand alone for any reasons
- Use a reference to the One-side in the N-side objects if the cardinality is one-to-squillions

Next time we'll see how to use two-way relationship and denormalizing to enhance the performance of these basic schemas.

6 Rules of Thumb for MongoDB Schema Design: Part 2

This is the second stop on our tour of modeling One-to-N relationships in MongoDB. Last time I covered the three basic schema designs: embedding, child-referencing, and parent-referencing. I also covered the two factors to consider when picking one of these designs:

- Will the entities on the "N" side of the One-to-N ever need to stand alone?

- What is the cardinality of the relationship: is it one-to-few; one-to-many; or one-to-squillions?

With these basic techniques under our belt, I can move on to covering more sophisticated schema designs, involving two-way referencing and denormalization.

Intermediate: Two-Way Referencing

If you want to get a little bit fancier, you can combine two techniques and include both styles of reference in your schema, having both references from the "one" side to the "many" side and references from the "many" side to the "one" side.

For an example, let's go back to that task-tracking system. There's a "people" collection holding Person documents, a "tasks" collection holding Task documents, and a One-to-N relationship from Person -> Task. The application will need to track all of the Tasks owned by a Person, so we will need to reference Person -> Task.

With the array of references to Task documents, a single Person document might look like this:

On the other hand, in some other contexts this application will display a list of Tasks (for example, all of the Tasks in a multi-person Project) and it will need to quickly find which Person is responsible for each Task. You can optimize this by putting an additional reference to the Person in the Task document.

This design has all of the advantages and disadvantages of the “One-to-Many” schema, but with some additions. Putting in the extra ‘owner’ reference into the Task document means that it’s quick and easy to find the Task’s owner, but it also means that if you need to reassign the task to another person, you need to perform **two** updates instead of just one. Specifically, you’ll have to update both the reference from the Person to the Task document, and the reference from the Task to the Person. (And to the relational gurus who are reading this – you’re right: using this schema design means that it is no longer possible to reassign a Task to a new Person with a single atomic update. This is OK for our task-tracking system: you need to consider if this works with your particular use case.)

Intermediate: Denormalizing With “One-To-Many” Relationships

Beyond just modeling the various flavors of relationships, you can also add denormalization into your schema. This can eliminate the need to perform the application-level join for certain cases, at the price of some additional complexity when performing updates. An example will help make this clear.

Denormalizing from Many -> One

For the parts example, you could denormalize the name of the part into the ‘parts[]’ array. For reference, here’s the version of the Product document without denormalization.

Denormalizing would mean that you don’t have to perform the application-level join when displaying all of the part names for the product, but you would have to perform that join if you needed any other information about a part.

While making it easier to get the part names, this would add just a bit of client-side work to the application-level join:

Denormalizing saves you a lookup of the denormalized data at the cost of a more expensive update: if you've denormalized the Part name into the Product document, then when you update the Part name you must also update every place it occurs in the 'products' collection.

Denormalizing only makes sense when there's a high ratio of reads to updates. If you'll be reading the denormalized data frequently, but updating it only rarely, it often makes sense to pay the price of slower updates – and more complex updates – in order to get more efficient queries. As updates become more frequent relative to queries, the savings from denormalization decrease.

For example: assume the part name changes infrequently, but the quantity on hand changes frequently. This means that while it makes sense to denormalize the part name into the Product document, it does not make sense to denormalize the quantity on hand.

Also note that if you denormalize a field, you lose the ability to perform atomic and isolated updates on that field. Just like with the two-way referencing example above, if you update the part name in the Part document, and then in the Product document, there will be a sub-second interval where the denormalized 'name' in the Product document will not reflect the new, updated value in the Part document.

Denormalizing from One -> Many

You can also denormalize fields from the "One" side into the "Many" side:

However, if you've denormalized the Product name into the Part document, then when you update the Product name you must also update every place it occurs in the 'parts' collection. This is likely to be a more expensive update, since you're updating multiple Parts instead of a single Product. As such, it's significantly **more** important to consider the read-to-write ratio when denormalizing in this way.

Intermediate: Denormalizing With "One-To-Squillions" Relationships

You can also denormalize the “one-to-squillions” example. This works in one of two ways: you can either put information about the “one” side (from the 'hosts’ document) into the “squillions” side (the log entries), or you can put summary information from the “squillions” side into the “one” side.

Here’s an example of denormalizing into the “squillions” side. I’m going to add the IP address of the host (from the ‘one’ side) into the individual log message:

Your query for the most recent messages from a particular IP address just got easier: it’s now just one query instead of two.

In fact, if there’s only a limited amount of information you want to store at the “one” side, you can denormalize it ALL into the “squillions” side and get rid of the “one” collection altogether:

On the other hand, you can also denormalize into the “one” side. Let’s say you want to keep the last 1000 messages from a host in the 'hosts’ document. You could use the $each / $slice functionality introduced in MongoDB 2.4 to keep that list sorted, and only retain the last 1000 messages:

The log messages get saved in the 'logmsg’ collection as well as in the denormalized list in the 'hosts’ document: that way the message isn’t lost when it ages out of the 'hosts.logmsgs’ array.

Note the use of the projection specification ({_id:1}) to prevent MongoDB from having to ship the entire ‘hosts’ document over the network. By telling MongoDB to only return the _id field, I reduce the network overhead down to just the few bytes that it takes to store that field (plus just a little bit more for the wire protocol overhead).

Just as with denormalizing in the “One-to-Many” case, you’ll want to consider the ratio of reads to updates. Denormalizing the log messages into the Host document makes sense only if log messages are infrequent relative to the number of times the application needs to look at all of the messages for a single host. This particular denormalization is a bad idea if you want to look at the data less frequently than you update it.

Recap

In this post, I've covered the additional choices that you have past the basics of embed, child-reference, or parent-reference.

- You can use bi-directional referencing if it optimizes your schema, and if you are willing to pay the price of not having atomic updates
- If you are referencing, you can denormalize data either from the "One" side into the "N" side, or from the "N" side into the "One" side

When deciding whether or not to denormalize, consider the following factors:

- You cannot perform an atomic update on denormalized data
- Denormalization only makes sense when you have a high read to write ratio

Next time, I'll give you some guidelines to pick and choose among all of these options.

6 Rules of Thumb for MongoDB Schema Design: Part 3

This is our final stop in this tour of modeling One-to-N relationships in MongoDB. In the first post, I covered the three basic ways to model a One-to-N relationship. Last time, I covered some extensions to those basics: two-way referencing and denormalization.

Denormalization allows you to avoid some application-level joins, at the expense of having more complex and expensive updates. Denormalizing one or more fields makes sense if those fields are read much more often than they are updated.

Read part one and part two if you've missed them.
Whoa! Look at All These Choices!

So, to recap:

- You can embed, reference from the "one" side, or reference from the "N" side, or combine a pair of these techniques

- You can denormalize as many fields as you like into the “one” side or the “N” side

  Denormalization, in particular, gives you a lot of choices: if there are 8 candidates for denormalization in a relationship, there are 2 ^{8} (1024) different ways to denormalize (including not denormalizing at all). Multiply that by the three different ways to do referencing, and you have over 3,000 different ways to model the relationship.

  Guess what? You now are stuck in the “paradox of choice” – because you have so many potential ways to model a “one-to-N” relationship, your choice on how to model it just got harder. Lots harder.

  Rules of Thumb: Your Guide Through the Rainbow

  Here are some “rules of thumb” to guide you through these in-denumerable (but not infinite) choices

- One: favor embedding unless there is a compelling reason not to
- Two: needing to access an object on its own is a compelling reason not to embed it
- Three: 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.
- Four: Don’t be afraid of application-level joins: if you index correctly and use the projection specifier (as shown in part 2) then application-level joins are barely more expensive than server-side joins in a relational database.
- Five: Consider the write/read ratio when denormalizing. A field that will mostly be read and only seldom updated is a good candidate for denormalization: if you denormalize a field that is updated frequently then the extra work of finding and updating all the instances is likely to overwhelm the savings that you get from denormalizing.
- Six: 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.
  Your Guide To The Rainbow

When modeling “One-to-N” relationships in MongoDB, you have a variety of choices, so you have to carefully think through the structure of your data. The main criteria you need to consider are:

- What is the cardinality of the relationship: is it “one-to-few”, “one-to-many”, or “one-to-squillions”?
- Do you need to access the object on the “N” side separately, or only in the context of the parent object?
- What is the ratio of updates to reads for a particular field?

Your main choices for structuring the data are:

- For “one-to-few”, you can use an array of embedded documents
- For “one-to-many”, or on occasions when the “N” side must stand alone, you should use an array of references. You can also use a “parent-reference” on the “N” side if it optimizes your data access pattern.
- For “one-to-squillions”, you should use a “parent-reference” in the document storing the “N” side.

Once you’ve decided on the overall structure of the data, then you can, if you choose, denormalize data across multiple documents, by either denormalizing data from the “One” side into the “N” side, or from the “N” side into the “One” side. You’d do this only for fields that are frequently read, get read much more often than they get updated, and where you don’t require strong consistency, since updating a denormalized value is slower, more expensive, and is not atomic.

Productivity and Flexibility

The upshot of all of this is that MongoDB gives you the ability to design your database schema to match the needs of your application. You can structure your data in MongoDB so that it adapts easily to change, and supports the queries and updates that you need to get the most out of your application.