By William Zola, Lead Technical Support Engineer at MongoDB

"I have lots of experience with SQL, but I'm just a beginner with MongoDB. How do I model a one-to-N relationship?" This is one of the more common questions I get from users attending MongoDB office hours.

I don't have a short answer to this question, because there isn't just one way, there's a whole rainbow's worth of ways. 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 mes 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).toAr
```

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.

- Part 2: Two-way referencing and denormalization
- Part 3: Your guide through the rainbow

More Information

- Schema Design Consulting Services
- Thinking in Documents (recorded webinar)
- Schema Design for Time-Series Data (recorded webinar)

- Socialite, the Open Source Status Feed - Storing a Social Graph (recorded webi

This post was updated in January 2015 to include additional resources and updated links.