By William Zola, Lead Technical Support Engineer at MongoDB

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 indenumberable (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 referenc
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.