By William Zola, Lead Technical Support Engineer at MongoDB

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:

```
db.person.findOne()
{
    _id: ObjectID("AAF1"),
    name: "Kate Monster",
    tasks [     // array of references to Task documents
        ObjectID("ADF9"),
        ObjectID("AE02"),
        ObjectID("AE73")
        // etc
    ]
}
```

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

```
db.tasks.findOne()
{
    _id: ObjectID("ADF9"),
    description: "Write lesson plan",
    due_date:  ISODate("2014-04-01"),
    owner: ObjectID("AAF1")     // Reference to Person 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 its 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.

```
> 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
    ]
}
```

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.

```
> db.products.findOne()
{
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [
        { id : ObjectID('AAAA'), name : '#4 grommet' },         // Part name is der
        { id: ObjectID('F17C'), name : 'fan blade assembly' },
        { id: ObjectID('D2AA'), name : 'power switch' },
        // etc
    ]
}
```

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

```
// Fetch the product document
> product = db.products.findOne({catalog_number: 1234});
  // Create an array of ObjectID()s containing *just* the part numbers
> part_ids = product.parts.map( function(doc) { return doc.id } );
  // Fetch all the Parts that are linked to this Product
> product_parts = db.parts.find({_id: { $in : part_ids } } ).toArray() ;
```

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 an 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 isc updates on that field. Just like with the two-way referencing example above, if you up 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:

```
> db.parts.findOne()
{
    _id : ObjectID('AAAA'),
    partno : '123-aff-456',
    name : '#4 grommet',
    product_name : 'left-handed smoke shifter',   // Denormalized from the 'Product
    product_catalog_number: 1234,                     // Ditto
    qty: 94,
    cost: 0.94,
    price: 3.99
}
```

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:

```
> db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    ipaddr : '127.66.66.66',
    host: ObjectID('AAAB')
}
```

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

```
> last_5k_msg = db.logmsg.find({ipaddr : '127.66.66.66'}).sort({time : -1}).limit(5
```

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:

```
> db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    ipaddr : '127.66.66.66',
    hostname : 'goofy.example.com',
}
```

On the other hand, you can also denormalize into the "one" side. Lets 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.

```
 //  Get log message from monitoring system
logmsg = get_log_msg();
log_message_here = logmsg.msg;
log_ip = logmsg.ipaddr;
  // Get current timestamp
now = new Date()
  // Find the _id for the host I'm updating
host_doc = db.hosts.findOne({ipaddr : log_ip },{_id:1});  // Don't return the whole
host_id = host_doc._id;
  // Insert the log message, the parent reference, and the denormalized data into t
db.logmsg.save({time : now, message : log_message_here, ipaddr : log_ip, host : hos
  // Push the denormalized log message onto the 'one' side
db.hosts.update( {_id: host_id },
        {$push : {logmsgs : { $each:  [ { time : now, message : log_message_here }
                           $sort:  { time : 1 },  // Only keep the latest ones
```

```
                                    $slice: -1000 }        // Only keep the latest 10
        }} );
```

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.

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 - Managing the Social Graph (recorded presentation)

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

Contact Us

Legal Notices

Security Information

Office Locations

Code of Conduct

Follow Us

- Facebook
- Github
- Youtube
- Twitter
- LinkedIn
- Slack
- StackOverflow

Get MongoDB Email Updates

Email Address →