6 Rules of Thumb for MongoDB Schema Design

William Zola
June 11, 2014 | Updated: November 2, 2022
#Atlas

“I have lots of experience with SQL and normalized databases, 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, In this post, I’ll talk about the three basic ways to model One-to-N relationships. I’ll also cover more sophisticated schema design including denormalization and two-way referencing. And I’ll review the entire range 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. Jump the end of the post for an explanation of what database denormalization is and when and when not to denormalize data.

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 and normalized tables: 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 retrieving data for this use case later in the 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 disadvar embedding. Each Part is a stand-alone document, so it's easy to search them a update them independently. One trade off for using this schema is having to pe second query to get details about the Parts for a Product. (But hold that thought until we get to denormalization.)

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-squillion

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 database schema for a normalized database. 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 reasons.
- Use a reference to the One-side in the N-side objects if the cardinality is one-to-squillions.

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 to Task. The application will need to track all of the Tasks owned by a Person, so we will need to reference Person to 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 (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 data retrieval for this purp


```
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 database gurus who are reading this, you're right; using this schema design over a normalized database model 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: Database denormalization 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.

Database denormalization from many to 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
    ]
}
```

Contact Us

Denormalization 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 denormalized
        { 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() ;
```

Denormalization saves you a lookup of the denormalized data at the cost of a more expensive update since you're adding a little data redundancy to the database: 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.

Denormalization 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 write performance–and more complex updates for redundant data–in order to get more efficient query performance. As updates become more frequent relative to queries, the savings from denormalization decreases.

For instance, 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, for data integrity purposes, 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 two-way referencing, if you update the part name in the Part document, and then in the Product document, data anomalies could occur since 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.

Database denormalization from one to many

You can also denormalize fields from the “ ” 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’ document
    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 to avoid data anomalies. 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: Database denormalization with one-to-squillions relationships

You can also denormalize the one-to-squillions relationship. 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 denormalization 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(5000).toArray()
```

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. Let's say you want to keep the last 1,000 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 1,000 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 document
host_id = host_doc._id;
  // Insert the log message, the parent reference, and the denormalized data into the 'many' side
db.logmsg.save({time : now, message : log_message_here, ipaddr : log_ip, host : host_id ) });
  // 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 1000
         }} );
```

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, you 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 protocol overhead).

Just as with denormalization in the “One-to-Many” case, you’ll want to consider the ratio of reads to updates. Denormalization of 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 section, 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 on database denormalization, 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.

Whoa! Looks at all these database denormalization choices!

Database Denormalization, in particular, gives you a lot of choices: if there are 8 candidates for denormalization in a relationship, there are 2 8 (1,024) different ways to denormalize (including no denormalization 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 a lot harder.

Database denormalization rules of thumb: Your guide through the rainbow

Here are some “rules of thumb” to guide you through these innumerable (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, then application-level joins are barely more expensive than server-side joins in a relational database.
- **Five:** Consider the read-to-write ratio with denormalization. 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 of redundant data is likely to overwhelm the savings that you get from denormalization.
- **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 “child-reference” in the document storing the “N” side.

Once you’ve decided on the overall structure of the data in your database design, 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.

Appendix I: What is database denormalization?

There is a very simple principle behind database denormalization techniques and it is this: Data that is accessed together should be stored together. Denormalization is the process of duplicating fields or deriving new fields from existing ones. Denormalized databases can improve read performance and query performance in a variety of cases, such as:

- A recurring query requires a few fields from a large document in another collection. You may choose to maintain a copy of those fields in an embedded document in the collection that the recurring query targets to avoid merging two distinct collections or performing frequent $lookup operations.
- An average value of some field in a collection is frequently requested. You may choose to create a derived field in a separate collection that is updated as your writes and maintains a running average for that field.

While embedding documents or arrays without data duplication is preferred for grouping related data, denormalization can improve read performance when separate collections must be maintained.

A single document can represent an entire customer order or the energy production for a day for a specific solar panel. Some users coming from the relational database who are more familiar with a normalized database model world treat the document as a row in a

table or spread across multiple tables. Wh[illegible]thing is stopping you from architecting your schema this way, it isn't the more e[illegible]y to store data or query large amounts of data, especially IoT data.

Denormalization enables you to increase performance of the database while having fewer joins compared with the normalized database model of a relational database.

Although MongoDB supports multi-document transactions for replica sets (starting in version 4.0) and sharded clusters (starting in version 4.2), for many scenarios, the denormalized database model will continue to be optimal for your data and use cases.

Note that with a denormalized database, it's important to maintain consistent duplicate data . However, in most cases, the increase in data retrieval performance and query execution will outweigh the presence of redundant copies of data and the need to avoid data inconsistency.

Appendix II: When does database denormalization make sense vs. database normalization?

Denormalization makes sense when you have a high read-to-write ratio. With denormalization you can avoid costly joins, at the expense of having more complex and expensive updates. Therefore, you should practice denormalization on only those fields that are read most often and are seldom updated since data redundancy is less of an issue.

$lookup operations join data from two collections in the same database based on a specified field. $lookup operations can be useful when your data is structured similarly to a relational database and you need to model large hierarchical datasets that wold normally be spread across multiple tables. However, these operations can be slow and resource-intensive because they need to read and perform logic on two collections instead of a single collection.

If you frequently run $lookup operations, consider restructuring your schema th[illegible] denormalization such that the your application can query a single collection to[illegible] of the information it needs. Use embedded documents and arrays to capture relationships between data in a single document structure. Use database denormalization to take advantage of MongoDB's rich document model, which allows your application to retrieve and manipulate related data in a single query execution.

Typically it is most advantageous to embrace database denormalization for operational databases–the efficiency of reading or writing an entire record in a single operation outweighing any modest increase in storage requirements.

Normalized data models describe relatio using references between documents. In general, use normalized data models in ing scenarios:

- When embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of data duplication.
- To represent more complex many-to-many relationships.
- To model large hierarchical data sets.

Today, there's a wide range of database design options in today's market. The relational database model and the practice of database normalization has its merits and limitations. The need to perform join operations across tables impedes performance, inhibits scaling, and introduces technical and cognitive overhead. Developers often create workarounds in their databases to achieve efficiency advantages. Those applications based on the relational databases that are performant will often incorporate ad hoc denormalization, materialized views, and external caching layers to get around the limitations of the normalized relational database.

More information

- Schema Design and Consulting Services
- Thinking in Documents (recorded webinar)

← Previous

Next →

Using MongoDB Backup to Start Over After a Corrupt or Lost Database

One of the most common scenarios for needing to go to a backup is human error. Someone releases...

June 10, 2014

Retrieval Augmented Generation for Claim Processing: Combining MongoDB Atlas Vector Search...

Following up on our previous blog, AI, Vectors, and the Future of Claims Processing: Why Insurance Needs to...

April 18, 2024

Social

GitHub

LinkedIn

Twitter

Facebook

Stack Overflow

YouTube

Twitch

Contact Us