Data Modeling at Scale: MongoDb + Mongoid, Callbacks, and Denormalizing Data for Efficiency

Aug 12 2011

I found myself confronted with a MongoDb data modeling problem. I have your vanilla User model which has many Items. The exact nature of an Item is irrelevant, but let us say a User can have lots of Items. I struggled with trying to figure out how to model this data in a flexible way while still leveraging the documented-orientated nature of MongoDb. The answer may seem obvious to some but it is interesting to weigh the options available.

To Embed or Not to Embed

The main choice was to embed Items in a User or have that as a separate collection. I do not think it makes sense to go vice versa, as Users are unique and clearly a top level entity. It would not make sense to have thousands of the same User in an Items collection. So the choice was between having Items in its own collection or embedding it in Users. A couple of factors came into play: How can I access, sort, or page through Item results if it is embedded in a User? What happens if I had so many Items in a User class I hit the MongoDb 4mb document size limit? (Unlikely: 4mb is a lot of data, but I would certainly not want to have to refactor that logic later on!) What would sharding look like with a large number of very large User documents? Most importantly, at what point would the number of Items be problematic with this approach? A

hundred? A thousand? A hundred thousand?

When to Embed

I think embedded documents are an awesome feature of MongoDb, and the general approach, as recommended on the docs, is to say "Why wouldn't I put this in an embedded document?". I would say if the number of Items a User would have is relatively small (say, enough that you would not need to page them on a UI, or if it would not create large network io by just accessing that field) then it can be an embedded document. The decision is a lot simpler if it is a 1..1 relationship as the potential size is clearly defined. 1..N relationships break down with embedded relations when N becomes so large that accessing it as a whole is impractical. As far as I know there does not seem to be a way to page or sort through an embedded array directly within MongoDb: you need to pull the entire field out of the database with field selection and then page on the client. Note MongoDb offers numerous ways to find data within a document no matter how it is stored within the document (see the docs on dot notation for more). You can even query on the position of elements in an array, which is helpful with sorted embedded lists (find me all Users who have Item Z as the first element). But sadly you cannot say "give me the first to the Nth element in an embedded array". It is all or nothing.

Now Mongoid does offer the ability to page through an embedded association using a gem (seems like people use Kaminari as will*paginate was removed from Mongoid some time ago). However, this paging is done within the ruby object for embedded relations. More importantly, it is only done on a per-document basis. Under the hood you need to grab the entire embedded relation _embedded within its root document* (think an array of Users containing an array of Items, not a plain array of Items). This means you cannot grab a collection of embedded documents which span

multiple root documents. You cannot say "give me all Items of type 'X'. You need to say "give me all Users and its Items containing Items of type 'X'". If you ever ran into the "Access to the collection for XXX is not allowed since it is an embedded document, please access a collection from the root document" error you are probably trying to issue an unsupported Mongoid query by bypassing a root document. You think you can treat embedded relations like normal collections, but you can't.

When to Have Separate Collections

So where does that leave us: If the relation is small enough, than an embedded relation is fine: we just need to realize that we can never really treat elements in that collection across its top level document and that getting those elements is an all-or-nothing decision for each parent document. For the sake of argument, let us say a User can have thousands of Items, and we wanted the ability to list Items across Users in a single view. That would be too much to manage as its own field as an embedded document, and we could not aggregate Items across Users easily. So it needs to be in its own collection. This now gives us numerous sorting options and paging features like skip and limit to reduce network traffic. If we have Items as its own collection then we can create a DBRef between the two. This is a classical relational breakdown. The thing that smells with this approach, specifically when using MongoDb, is that if I were viewing a list of Items, and wanted to show the username associated with them, I would either have to use a DBRef command to pull user information or make two queries. Less than ideal. A JOIN would certainly be easier (albeit at scale, impractical, but probably for the DbRef approach too).

The Solution

So what I'm really looking for is the ability to show the

username with a list of Items when each has its own collection. The trick is I do not need to aggregate this data when I am pulling it out of the database. Instead I can assemble it before I put in the database and it will all be there when I take it out. Classic denormalization. With Mongoid and Callbacks this becomes extremely easy.

On my Items class I add a *:belongs_to :user* property along with a *:username* property. I want to ensure that a *:user* always exists, so I add a *validates_presence_of :user* validation. I do not need to add *:username* to this validation as we will see below. Then I leverage callbacks like so:

```
before_save :add_username

protected
def add_username
  if user_id_changed?
    self.username = user.username
  end
end
```

What will happen is if the User property changed Mongoid will set the current Item's username value to the user.username property value. The username field is now stored within the Item document, and I can query on this field as easily as any other Item property (including the user_id relation on the Item document). More importantly, it is already available in a query result so there is no need to make an additional query on User.username for display. Any time the user changes (if Items can switch Users) the username will be updated automatically before the save to maintain consistency. Because the :user object is required, there is no need to also make :username required. Username will read from the required User property before each save. There is a slight catch with this approach: callbacks will only be run on document which received the save call, so be careful

with cascading updates. As always a great test suite will always ensure the behavior you want is enforced.

Sharding

The other point about the user relation, whether it is via the username field or on user_id, is that it makes a good shard key. If we shard off of this field (probably in conjunction of another key) you can control things like write scaling while keeping relevant data close together for querying. For instance, sharding only on username will put all data in the same server to make querying a user's items extremely efficient. Sharding on username and something else will get writes distributed across servers at the expense of having to gather elements across servers when returning results. The bottom line is know your use case: are faster writes more important than faster reads? Which one are you doing more of?

In Conclusion

I think there are two important things to realize when it comes to modeling with not just Mongoid but with any type of data store, sql or nosql. First when you are dealing with scale you want to put your data in the same way you want to get it out. Know your data access patterns. Sql allows a tremendous amount of flexibility, but joining numerous tables across millions of rows is extremely inefficient. More importantly, if you model your data in NoSql incorrectly, you could end up with similar performance problems. In the case of the data denormalization exercise above, adding a username field to the Items collections saves us from a DbRef later. Plus, with the use of callbacks, getting our data into Mongoid in a denormalized way is easy. We could easily apply the same principle to a sql-based solution: add a username column to a Item table or create a materialized view/indexed view on the Users/Items data. If you are debating a no-sql solution over a sql one, take a look at

the cost/benefit of one approach over another in terms of how easy it is to model your data around data access. I think MongoDb gives a good amount of flexibility, especially with querying and indices, while still promoting some of the NoSql goodies like easy sharding for scalability and easy replication for reliability and read scaling.

Secondly, it is extremely important to know your toolset. With MongoDb, you get a tremendous amount of querying power: filtering on any field, no matter the nesting, even if it's an array; creating indices on said fields; map/reduce views; only retrieving specific fields from a document; the list is nearly endless. ORM features are important too: How does Mongoid map its API to MongoDB commands? How does it deal with dirty tracking? What callbacks are available? The coolest thing on the Mongoid website is the statement *This is why the documentation provides the exact queries that Mongoid is executing against the database when you call a persistence operation. If we took the time to tell you, you should listen.* VERY TRUE! I like that. The point being, there should be a purpose why you are choosing a NoSql solution: so know what it is and leverage it. It will mean the difference of succeeding at scale or failing at launch.

Cassandra

As an interesting footnote, I think Cassandra exemplifies the query-first approach to data modeling (I mean, it states so on its wiki!). Cassandra's uniqueness is in its masterless approach as a key/value store. It comes with some interesting features: the choice of using a secondary index vs. columnfamily as index, numerous comparison operators on columnfamily names, super columns vs. columns for storing data, replication and write consistency options across multiple data centers. This leads to plenty of benefits but with a certain cost. As for the know your tools/know your data philosophy, an

example is the typical choice of "Do you create a row and use its respected columns as an index, choosing an appropriate column comparison type, or do you treat your data as a key/value store and use a secondary index for queries?" One the one hand, you have a pre-sorted list that queries from one machine and with one call with slices for paging; on the other, you may need to farm out to a lot of machines to get the data you want. Knowing your options is important, and knowing what you have to do to implement your choice is nearly as important. Even with the best Cassandra ORMs you still need to do a lot of prep to get your data into and out of Cassandra in a meaningful way.

Final Thought

In a bit of contradictory advice, I'd say don't sweat it too much. Do some preliminary research, go with your hunch and trust your ability to refactor when needed. If you wait to figure out the perfect solution, you won't build anything!