No Joins

The first and most fundamental difference that you'll need to get comfortable with is MongoDB's lack of joins. I don't

know the specific reason why some type of join syntax isn't supported in MongoDB, but I do know that joins are

generally seen as non-scalable. That is, once you start to split your data horizontally, you end up performing your joins

on the client (the application server) anyway. Regardless of the reasons, the fact remains that data *is* relational, and

MongoDB doesn't support joins.

Without knowing anything else, to live in a join-less world, we have to do joins ourselves within our application's code.

Essentially we need to issue a second query to find the relevant data in a second collection. Setting our data up isn't

any different than declaring a foreign key in a relational database. Let's give a little less focus to our beautiful unicorns

and a bit more time to our employees. The first thing we'll do is create an employee (I'm providing an explicit _id so

that we can build coherent examples)

Arrays and Embedded Documents

Just because MongoDB doesn't have joins doesn't mean it doesn't have a few tricks up its sleeve. Remember when

we saw that MongoDB supports arrays as first class objects of a document? It turns out that this is incredibly handy

when dealing with many-to-one or many-to-many relationships

Denormalization

Yet another alternative to using joins is to denormalize your data. Historically, denormalization was reserved for

performance-sensitive code, or when data should be snapshotted (like in an audit log). However, with the evergrowing

popularity of NoSQL, many of which don't have joins, denormalization as part of normal modeling is becoming

increasingly common. This doesn't mean you should duplicate every piece of information in every document. However,

rather than letting fear of duplicate data drive your design decisions, consider modeling your data based on what

information belongs to what document.

For example, say you are writing a forum application. The traditional way to associate a specific user with a post is

via a userid column within posts. With such a model, you can't display posts without retrieving (joining to) users.

A possible alternative is simply to store the name as well as the userid with each post. You could even do so with an

embedded document, like user: {id: ObjectId('Something'), name: 'Leto'}. Yes, if you let users change their

name, you may have to update each document (which is one multi-update).

Adjusting to this kind of approach won't come easy to some. In a lot of cases it won't even make sense to do this.

Don't be afraid to experiment with this approach though. It's not only suitable in some circumstances, but it can also

be the best way to do it.