# Modeling data in DocumentDB

While schema-free databases, like DocumentDB, make it super easy to embrace changes to your data model you should still spend some time thinking about your data.

How is data going to be stored? How is your application going to retrieve and query data? Is your application read heavy, or write heavy?

After reading this article, you will be able to answer the following questions:

* How should I think about a document in a document database?
* What is data modeling and why should I care?
* How is modeling data in a document database different to a relational database?
* How do I express data relationships in a non-relational database?
* When do I embed data and when do I link to data?

## Embedding data

When you start modeling data in a document store, such as DocumentDB, try to treat your entities as **self-contained documents** represented in JSON.

Before we dive in too much further, let us take a few steps back and have a look at how we might model something in a relational database, a subject many of us are already familiar with. The following example shows how a person might be stored in a relational database.



Relational database model

When working with relational databases, we’ve been taught for years to normalize, normalize, normalize.

Normalizing your data typically involves taking an entity, such as a person, and breaking it down in to discreet pieces of data. In the example above, a person can have multiple contact detail records as well as multiple address records. We even go one step further and break down contact details by further extracting common fields like a type. Same for address, each record here has a type like *Home* or *Business*

The guiding premise when normalizing data is to **avoid storing redundant data** on each record and rather refer to data. In this example, to read a person, with all their contact details and addresses, you need to use JOINS to effectively aggregate your data at run time.

SELECT p.FirstName, p.LastName, a.City, cd.Detail  
FROM Person p  
JOIN ContactDetail cd ON cd.PersonId = p.Id  
JOIN ContactDetailType on cdt ON cdt.Id = cd.TypeId  
JOIN Address a ON a.PersonId = p.Id

Updating a single person with their contact details and addresses requires write operations across many individual tables.

Now let’s take a look at how we would model the same data as a self-contained entity in a document database.

{  
 "id": "1",  
 "firstName": "Thomas",  
 "lastName": "Andersen",  
 "addresses": [  
 {   
 "line1": "100 Some Street",  
 "line2": "Unit 1",  
 "city": "Seattle",  
 "state": "WA",  
 "zip": 98012  
 }  
 ],  
 "contactDetails": [  
 {"email: "thomas@andersen.com"},  
 {"phone": "+1 555 555-5555", "extension": 5555}  
 ]   
}

Using the approach above we have now **denormalized** the person record where we **embedded** all the information relating to this person, such as their contact details and addresses, in to a single JSON document. In addition, because we’re not confined to a fixed schema we have the flexibility to do things like having contact details of different shapes entirely.

Retrieving a complete person record from the database is now a single read operation against a single collection and for a single document. Updating a person record, with their contact details and addresses, is also a single write operation against a single document.

By denormalizing data, your application may need to issue fewer queries and updates to complete common operations.

### When to embed

In general, use embedded data models when:

* There are **contains** relationships between entities.
* There are **one-to-few** relationships between entities.
* There is embedded data that **changes infrequently**.
* There is embedded data won’t grow **without bound**.
* There is embedded data that is **integral** to data in a document.

[AZURE.NOTE] Typically denormalized data models provide better **read** performance.

### When not to embed

While the rule of thumb in a document database is to denormalize everything and embed all data in to a single document, this can lead to some situations that should be avoided.

Take this JSON snippet.

{  
 "id": "1",  
 "name": "What's new in the coolest Cloud",  
 "summary": "A blog post by someone real famous",  
 "comments": [  
 {"id": 1, "author": "anon", "comment": "something useful, I'm sure"},  
 {"id": 2, "author": "bob", "comment": "wisdom from the interwebs"},  
 …  
 {"id": 100001, "author": "jane", "comment": "and on we go ..."},  
 …  
 {"id": 1000000001, "author": "angry", "comment": "blah angry blah angry"},  
 …  
 {"id": ∞ + 1, "author": "bored", "comment": "oh man, will this ever end?"},  
 ]  
}

This might be what a post entity with embedded comments would look like if we were modeling a typical blog, or CMS, system. The problem with this example is that the comments array is **unbounded**, meaning that there is no (practical) limit to the number of comments any single post can have. This will become a problem as the size of the document could grow significantly.

[AZURE.TIP] Documents in DocumentDB have a maximum size. For more on this refer to [DocumentDB limits](/documentation/articles/documentdb-limits).

As the size of the document grows the ability to transmit the data over the wire as well as reading and updating the document, at scale, will be impacted.

In this case it would be better to consider the following model.

Post document:  
{  
 "id": 1,  
 "name": "What's new in the coolest Cloud",  
 "summary": "A blog post by someone real famous",  
 "recentComments": [  
 {"id": 1, "author": "anon", "comment": "something useful, I'm sure"},  
 {"id": 2, "author": "bob", "comment": "wisdom from the interwebs"},  
 {"id": 3, "author": "jane", "comment": "....."}  
 ]  
}  
  
Comment documents:  
{  
 "postId": 1  
 "comments": [  
 {"id": 4, "author": "anon", "comment": "more goodness"},  
 {"id": 5, "author": "bob", "comment": "tails from the field"},  
 ...  
 {"id": 99, "author": "angry", "comment": "blah angry blah angry"}  
 ]  
},  
{  
 "postId": 1  
 "comments": [  
 {"id": 100, "author": "anon", "comment": "yet more"},  
 ...  
 {"id": 199, "author": "bored", "comment": "will this ever end?"}  
 ]  
}

This model has the three most recent comments embedded on the post itself, which is an array with a fixed bound this time. The other comments are grouped in to batches of 100 comments and stored in separate documents. The size of the batch was chosen as 100 because our fictitious application allows the user to load 100 comments at a time.

Another case where embedding data is not a good idea is when the embedded data is used often across documents and will change frequently.

Take this JSON snippet.

{  
 "id": "1",  
 "firstName": "Thomas",  
 "lastName": "Andersen",  
 "holdings": [  
 {  
 "numberHeld": 100,  
 "stock": { "symbol": "zaza", "open": 1, "high": 2, "low": 0.5 }  
 },  
 {  
 "numberHeld": 50,  
 "stock": { "symbol": "xcxc", "open": 89, "high": 93.24, "low": 88.87 }  
 }  
 ]  
}

This could represent a person’s stock portfolio. We have chosen to embed the stock information in to each portfolio document. In an environment where related data is changing frequently, like a stock trading application, embedding data that changes frequently is going to mean that you are constantly updating each portfolio document every time a stock is traded.

Stock *zaza* may be traded many hundreds of times in a single day and thousands of users could have *zaza* on their portfolio. With a data model like the above we would have to update many thousands of portfolio documents many times every day leading to a system that won’t scale very well.

## Referencing data

So, embedding data works nicely for many cases but it is clear that there are scenarios when denormalizing your data will cause more problems than it is worth. So what do we do now?

Relational databases are not the only place where you can create relationships between entities. In a document database you can have information in one document that actually relates to data in other documents. Now, I am not advocating for even one minute that we build systems that would be better suited to a relational database in DocumentDB, or any other document database, but simple relationships are fine and can be very useful.

In the JSON below we chose to use the example of a stock portfolio from earlier but this time we refer to the stock item on the portfolio instead of embedding it. This way, when the stock item changes frequently throughout the day the only document that needs to be updated is the single stock document.

Person document:  
{  
 "id": "1",  
 "firstName": "Thomas",  
 "lastName": "Andersen",  
 "holdings": [  
 { "numberHeld": 100, "stockId": 1},  
 { "numberHeld": 50, "stockId": 2}  
 ]  
}  
  
Stock documents:  
{  
 "id": 1,  
 "symbol": "zaza",  
 "open": 1,  
 "high": 2,  
 "low": 0.5,  
 "vol": 11970000,  
 "mkt-cap": 42000000,  
 "pe": 5.89  
},  
{  
 "id": 2,  
 "symbol": "xcxc",  
 "open": 89,  
 "high": 93.24,  
 "low": 88.87,  
 "vol": 2970200,  
 "mkt-cap": 1005000,  
 "pe": 75.82  
}

An immediate downside to this approach though is if your application is required to show information about each stock that is held when displaying a person’s portfolio; in this case you would need to make multiple trips to the database to load the information for each stock document. Here we’ve made a decision to improve the efficiency of write operations, which happen frequently throughout the day, but in turn compromised on the read operations that potentially have less impact on the performance of this particular system.

[AZURE.NOTE] Normalized data models **can require more round trips** to the server.

### What about foreign keys?

Because there is currently no concept of a constraint, foreign-key or otherwise, any inter-document relationships that you have in documents are effectively “weak links” and will not be verified by the database itself. If you want to ensure that the data a document is referring to actually exists, then you need to do this in your application, or through the use of server-side triggers or stored procedures on DocumentDB.

### When to reference

In general, use normalized data models when:

* Representing **one-to-many** relationships.
* Representing **many-to-many** relationships.
* Related data **changes frequently**.
* Referenced data could be **unbounded**.

[AZURE.NOTE] Typically normalizing provides better **write** performance.

### Where do I put the relationship?

The growth of the relationship will help determine in which document to store the reference.

If we look at the JSON below that models publishers and books.

Publisher document:  
{  
 "id": "mspress",  
 "name": "Microsoft Press",  
 "books": [ 1, 2, 3, ..., 100, ..., 1000]  
}  
  
Book documents:  
{"id": 1, "name": "DocumentDB 101" }  
{"id": 2, "name": "DocumentDB for RDBMS Users" }  
{"id": 3, "name": "Taking over the world one JSON doc at a time" }  
...  
{"id": 100, "name": "Learn about Azure DocumentDB" }  
...  
{"id": 1000, "name": "Deep Dive in to DocumentDB" }

If the number of the books per publisher is small with limited growth, then storing the book reference inside the publisher document may be useful. However, if the number of books per publisher is unbounded, then this data model would lead to mutable, growing arrays, as in the example publisher document above.

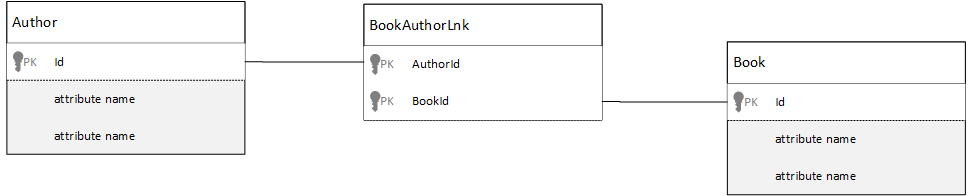
Switching things around a bit would result in a model that still represents the same data but now avoids these large mutable collections.

Publisher document:   
{  
 "id": "mspress",  
 "name": "Microsoft Press"  
}  
  
Book documents:   
{"id": 1,"name": "DocumentDB 101", "pub-id": "mspress"}  
{"id": 2,"name": "DocumentDB for RDBMS Users", "pub-id": "mspress"}  
{"id": 3,"name": "Taking over the world one JSON doc at a time"}  
...  
{"id": 100,"name": "Learn about Azure DocumentDB", "pub-id": "mspress"}  
...  
{"id": 1000,"name": "Deep Dive in to DocumentDB", "pub-id": "mspress"}

In the above example, we have dropped the unbounded collection on the publisher document. Instead we just have a a reference to the publisher on each book document.

### How do I model many:many relationships?

In a relational database *many:many* relationships are often modeled with join tables, which just join records from other tables together.



Join tables

You might be tempted to replicate the same thing using documents and produce a data model that looks similar to the following.

Author documents:   
{"id": 1, "name": "Thomas Andersen" }  
{"id": 2, "name": "William Wakefield" }  
  
Book documents:  
{"id": 1, "name": "DocumentDB 101" }  
{"id": 2, "name": "DocumentDB for RDBMS Users" }  
{"id": 3, "name": "Taking over the world one JSON doc at a time" }  
{"id": 4, "name": "Learn about Azure DocumentDB" }  
{"id": 5, "name": "Deep Dive in to DocumentDB" }  
  
Joining documents:   
{"authorId": 1, "bookId": 1 }  
{"authorId": 2, "bookId": 1 }  
{"authorId": 1, "bookId": 2 }  
{"authorId": 1, "bookId": 3 }

This would work. However, loading either an author with their books, or loading a book with its author, would always require at least two additional queries against the database. One query to the joining document and then another query to fetch the actual document being joined.

If all this join table is doing is gluing together two pieces of data, then why not drop it completely? Consider the following.

Author documents:  
{"id": 1, "name": "Thomas Andersen", "books": [1, 2, 3]}  
{"id": 2, "name": "William Wakefield", "books": [1, 4]}  
  
Book documents:   
{"id": 1, "name": "DocumentDB 101", "authors": [1, 2]}  
{"id": 2, "name": "DocumentDB for RDBMS Users", "authors": [1]}  
{"id": 3, "name": "Learn about Azure DocumentDB", "authors": [1]}  
{"id": 4, "name": "Deep Dive in to DocumentDB", "authors": [2]}

Now, if I had an author, I immediately know which books they have written, and conversely if I had a book document loaded I would know the ids of the author(s). This saves that intermediary query against the join table reducing the number of server round trips your application has to make.

## Hybrid data models

We’ve now looked embedding (or denormalizing) and referencing (or normalizing) data, each have their upsides and each have compromises as we have seen.

It doesn’t always have to be either or, don’t be scared to mix things up a little.

Based on your application’s specific usage patterns and workloads there may be cases where mixing embedded and referenced data makes sense and could lead to simpler application logic with fewer server round trips while still maintaining a good level of performance.

Consider the following JSON.

Author documents:   
{  
 "id": 1,  
 "firstName": "Thomas",  
 "lastName": "Andersen",   
 "countOfBooks": 3,  
 "books": [1, 2, 3],  
 "images": [  
 {"thumbnail": "http://....png"}  
 {"profile": "http://....png"}  
 {"large": "http://....png"}  
 ]  
},  
{  
 "id": 2,  
 "firstName": "William",  
 "lastName": "Wakefield",  
 "countOfBooks": 1,  
 "books": [1, 4, 5],  
 "images": [  
 {"thumbnail": "http://....png"}  
 ]  
}  
  
Book documents:  
{  
 "id": 1,  
 "name": "DocumentDB 101",  
 "authors": [  
 {"id": 1, "name": "Thomas Andersen", "thumbnailUrl": "http://....png"},  
 {"id": 2, "name": "William Wakefield", "thumbnailUrl": "http://....png"}  
 ]  
},  
{  
 "id": 2,  
 "name": "DocumentDB for RDBMS Users",  
 "authors": [  
 {"id": 1, "name": "Thomas Andersen", "thumbnailUrl": "http://....png"},  
 ]  
}

Here we’ve (mostly) followed the embedded model, where data from other entities are embedded in the top-level document, but other data is referenced.

If you look at the book document, we can see a few interesting fields when we look at the array of authors. There is an *id* field which is the field we use to refer back to an author document, standard practice in a normalized model, but then we also have *name* and *thumbnailUrl*. We could’ve just stuck with *id* and left the application to get any additional information it needed from the respective author document using the “link”, but because our application displays the author’s name and a thumbnail picture with every book displayed we can save a round trip to the server per book in a list by denormalizing **some** data from the author.

Sure, if the author’s name changed or they wanted to update their photo we’d have to go an update every book they ever published but for our application, based on the assumption that authors don’t change their names very often, this is an acceptable design decision.

In the example there are **pre-calculated aggregates** values to save expensive processing on a read operation. In the example, some of the data embedded in the author document is data that is calculated at run-time. Every time a new book is published, a book document is created **and** the countOfBooks field is set to a calculated value based on the number of book documents that exist for a particular author. This optimization would be good in read heavy systems where we can afford to do computations on writes in order to optimize reads.

The ability to have a model with pre-calculated fields is made possible because DocumentDB supports **multi-document transactions**. Many NoSQL stores cannot do transactions across documents and therefore advocate design decisions, such as “always embed everything”, due to this limitation. With DocumentDB, you can use server-side triggers, or stored procedures, that insert books and update authors all within an ACID transaction. Now you don’t **have** to embed everything in to one document just to be sure that your data remains consistent.

## Next steps

The biggest takeaways from this article is to understand that data modeling in a schema-free world is just as important as ever.

Just as there is no single way to represent a piece of data on a screen, there is no single way to model your data. You need to understand your application and how it will produce, consume, and process the data. Then, by applying some of the guidelines presented here you can set about creating a model that addresses the immediate needs of your application. When your applications need to change, you can leverage the flexibility of a schema-free database to embrace that change and evolve your data model easily.

To learn more about Azure DocumentDB, refer to the service’s [documentation](https://azure.microsoft.com/documentation/services/documentdb/) page.

To learn about tuning indexes in Azure DocumentDB, refer to the article on [indexing policies](/documentation/articles/documentdb-indexing-policies).

To understand how to shard your data across multiple partitions, refer to [Partitioning Data in DocumentDB](/documentation/articles/documentdb-partition-data).

And finally, for guidance on modeling data and sharding for multi-tenant applications, consult [Scaling a Multi-Tenant Application with Azure DocumentDB](http://blogs.msdn.com/b/documentdb/archive/2014/12/03/scaling-a-multi-tenant-application-with-azure-documentdb.aspx).