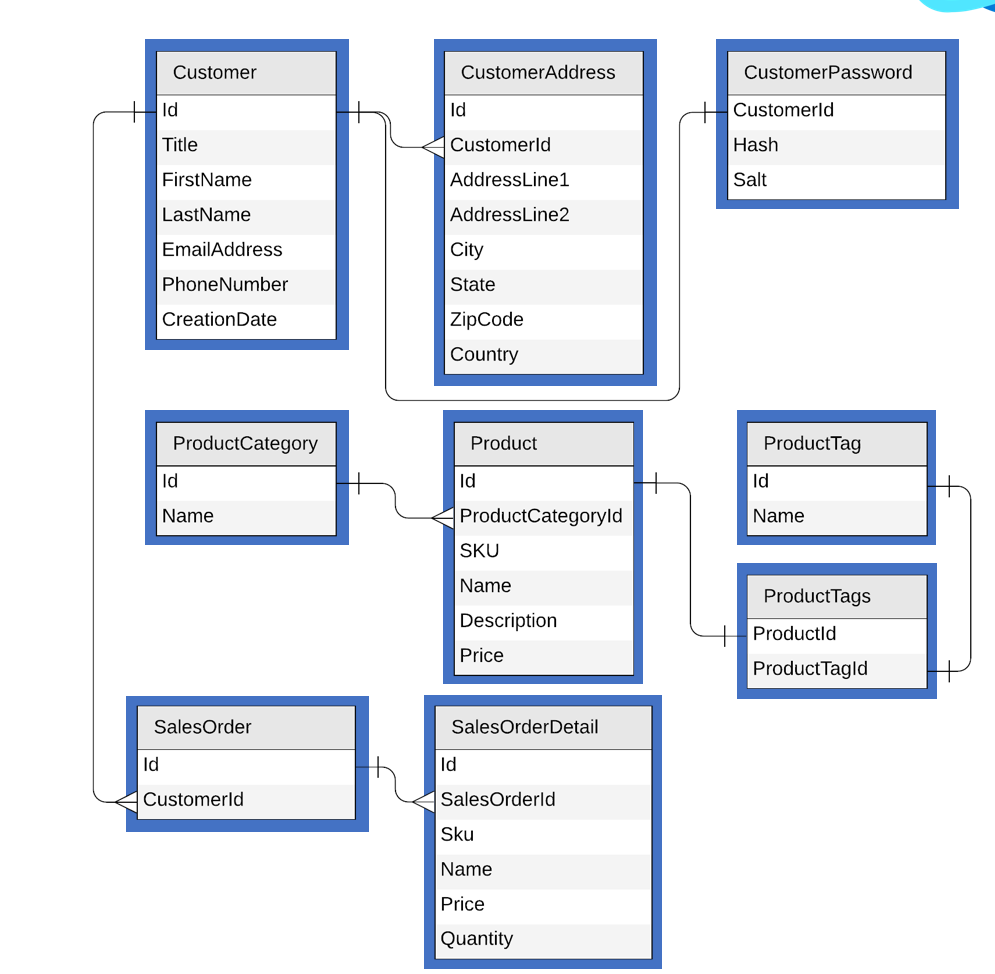
**WebStore relational model**

Our database is for an e-commerce web site that we’re calling WebStore. Here is the relational data model for WebStore in SQL Server:



This data model is relatively small, but still representative of a typical production model. It has one-to-many relationships, like the ones from Customer to CustomerAddress and SalesOrder. There is also a one-to-one relationship from Customer to CustomerPassword, and the ProductTags table implements a many-to-many relationship between Product and ProductTag.

### Container per table?

It’s very natural at first to think of a container in Azure Cosmos DB like a table. Your first instinct may be to say, OK, we have nine tables in our data model, let’s create nine containers in Azure Cosmos DB.

Now you can certainly do this, but again, this would be the worst possible design, being horizontally scalable and non-relational. Azure Cosmos DB will expose no way to join documents, or to enforce relational constraints between them. Therefore, this approach will not only perform poorly, but it will be very difficult to maintain and program against.

What’s the answer? Let’s get there one step at a time.

### Embed vs. reference

Let’s start with customers and their related entities. JSON is hierarchical, so we don’t need separate documents for every type. So now think about the distinction between one-to-many and one-to-few. It’s reasonable to impose an upper limit on the number of addresses a customer can have, and there’s only one password per customer, so we could combine all of those into a single document:

[](https://devblogs.microsoft.com/cosmosdb/wp-content/uploads/sites/52/2020/06/DM05-168x300.png)

This has immediately solved the problem of joining between customers and their addresses, and passwords, because that’s all “pre-joined” by embedding the one-to-few relationship for addresses as a nested array, and the one-to-one relationship for the password as an embedded object. Simply by embedding, we’ve reduced three relational tables to a single customer document.

On the other hand, we would certainly not want an upper limit on the number of sales orders per customer – ideally, that is unbounded (while the maximum document size is 2 MB). The orders will be stored in separate documents, referenced by customer ID.

The rules for when to embed and when to reference are simple, as we’re demonstrating. One-to-few and one-to-one relationships often benefit from embedding, while one-to-many (particularly unbounded) and many-to-many relationships require that you reference. Embedding is also useful when all the entities are typically queried and/or updated together (for example, a customer profile with addresses and password), while it’s usually better to separate entities that are most often queried and updated separately (such as individual sales orders).

The next – and arguably most important – step is to choose a partition key for the customer document. Making the right choice requires that you understand how partitioning works.

### Understanding partitioning

When you create a container, you supply a partition key. This is some value in your documents that Azure Cosmos DB groups documents together by in logical partitions. Each server in the cluster is a physical partition that can host any number of logical partitions, each of which in turn stores any number of documents with the same partition key value. Again, we don’t think about the physical partitions, we’re concerned primarily with the logical partitions that are based on the partition key that we select.

All the documents in a logical partition will always be stored on the same physical partition; a logical partition will never be spread across multiple servers in the cluster. Ideally, therefore, you want to choose a partition key whose value will be known for most of your typical queries. When the partition key is known, then Azure Cosmos DB can route your query directly to the physical partition where it knows all the documents that can possibly satisfy the query are stored. This is called a single-partition query.

If the partition key is not known, then it’s a cross-partition query (also often called a fan-out query). In this case, Azure Cosmos DB needs to visit every physical partition and aggregate their results into a single resultset for the query. This is fine for occasional queries, but adds unacceptable overhead for common queries in a heavy workload.

You also want a partition key that results in a uniform distribution of both storage and throughput. A logical partition can’t exceed 20 GB, but regardless, you don’t want some logical partitions to be huge and others very small. And from a throughput perspective, you don’t want some logical partitions to be heavily accessed for reads and writes, and not others. These “hot partition” situations should always be avoided.

### Choosing a partition key

With this understanding, we can select a partition key for the customer documents that we’ll store in a customer container. The question is always the same: “What’s the most common query?” For customers, we most often want to query a customer by its ID, like so:

SELECT \* FROM c WHERE c.id = ‘<custId>’

In this case, we want to choose the id property itself as the partition key. This means you’ll get only one document in every logical partition, which is fine. It’s desirable to use a partition key that yields a large spectrum of distinct values. You may have thousands of logical partitions for thousands of customers, but with only one document each, you will achieve a highly uniform distribution across the physical partitions.

We’ll take a very different approach for product categories. Users visiting the website will typically want to view the complete list of product categories. Then, they’ll want to query for all the product that belong to a category that interests them, which is essentially a query on the product category container with no WHERE clause. The problem though, is that would be a cross-partition query, and we want to get all our category documents using a single-partition query.

The trick here is to add another property called type to each product category document, and set its value to “category” in every document. Then we can partition the product category container on the type property. This would store all the category documents in a single logical partition, and the following query could retrieve them as a single-partition query:

SELECT \* FROM c WHERE c.type = ‘category’

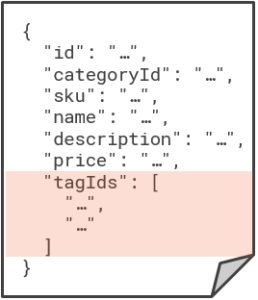
This same is true of tags; users will typically want a full list of tags and then drill to view the products associated with interesting tags. This is a typical pattern for short lookup lists that are often retrieved all at once. So that would be another container for product tags, partitioned on a type property where all the documents have the same value “tag” in that property, and then queried with:

SELECT \* FROM c WHERE c.type = ‘tag’

### Many-to-many relationships

Now for the many-to-many relationship between products and tags. This can be modeled by embedding an array of IDs on one side or the other; we could either store a list of tag IDs in each product, or a list of product IDs in each tag. Since there will be fewer products per tag than tags per product, we’ll store tag IDs in each product document, like so:

Once the user chooses a category, the next typical query would be to retrieve all the products in a given category by category ID, like so:

[](https://devblogs.microsoft.com/cosmosdb/wp-content/uploads/sites/52/2020/06/DM06-258x300.png)

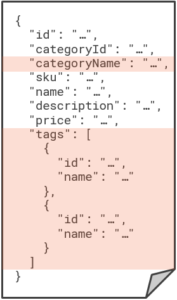
SELECT \* FROM c WHERE c.categoryId = ‘<catId>’

To make this a single-partition query, we want to partition the product contain on the product category ID, and that will store all the products for the same category in the same logical partition.

### Introduction denormalization

Now we have a new challenge, because product documents hold just the category ID and an array of tag IDs – it doesn’t have the category and tag names themselves. And we already know that Azure Cosmos DB won’t join related documents together for us. So, if we want to display the category and tag names on the web page – which we do – then we need to run additional queries to get that information. First, we need to query the product category container to get the category name, and then – for each product in the category – we need to query the product tag container to get all the tag names.

We definitely need to avoid this, and we’ll solve the problem using denormalization. And that just means that – unlike in a normalized data model – we will duplicate information as necessary in order to make it more readily available for queries that need it. That means that we’ll store a copy of the category name, and copies of the tag names, in each related product document:

[](https://devblogs.microsoft.com/cosmosdb/wp-content/uploads/sites/52/2020/06/DM07-177x300.png)

Now we have everything we need to display about a product self-contained inside a single product document. And that will work great, until of course, there’s a category name or tag name is changed. Because now we need a way to cascade that name change to all the related copies in order to keep our data consistent.

### Denormalizing with the change feed

This is a perfect situation for the Azure Cosmos DB change feed, which is a persistent log of all changes made to any container. By subscribing to the change feed on the category and tag containers, we can respond to updates and then propagate the change out to all related product documents so that everything remains in sync.

This can be achieved with a minimal amount of code, and implemented out-of-band with the main application by deploying the change feed code to run as an Azure function:

[](https://devblogs.microsoft.com/cosmosdb/wp-content/uploads/sites/52/2020/06/DM08-300x107.png)

Any change to a category or tag name triggers and Azure function to update all related product documents. This lets us maintain a denormalized model that’s optimized to retrieve all relevant information about a product with one single-partition query.

### Combining different types

The last part of our schema are the customer orders and order details. First, we’ll embed the details into each order as a single document for the sales order container, because that’s another one-to-few relationship between entities that are typically retrieved and updated together.

It will be very common for customers to retrieve their orders using the following query:

SELECT \* FROM c WHERE c.customerId = ‘<custId>’

That makes the customer ID the best choice for the partition key. But before we jump to create another container for sales orders, remember that we’re also partitioning customers on the customer ID in the customer container. And unlike a relational database where tables have defined schemas, Azure Cosmos DB lets you mix different types of documents in the same container. And it makes sense to do that when those different types share the same partition key.

We’ll combine customer and sales order documents in the same customer container, which will require just a minor tweak to the customer document. We’ll need to add a customerId property to hold a copy of the customer ID in the id property. Then we can partition on customerId which will be present in both document types:

[](https://devblogs.microsoft.com/cosmosdb/wp-content/uploads/sites/52/2020/06/DM09-300x124.png)

Notice that we’ve also added a type property to distinguish between the two types of documents. So now, there is still only one customer document per logical partition, but each logical partition also includes the related orders for that customer. And this kind of gets us our joins back, because now we can retrieve a customer and all their related orders with the following single-partition query:

SELECT \* FROM c WHERE c.id = ‘<custId>’

### Denormalizing with a stored procedure

Let’s wrap up with one more query to retrieve our top customers; essentially, a list of customers sorted descending by order count. In the relational world, we would just run a SELECT COUNT(\*) on the Order table with a GROUP BY on the customer ID, and then sort descending on that count.

But in Azure Cosmos DB, the answer is once again to denormalize. We’ll just add a salesOrderCount property to each customer document. Then our query becomes as simple as:

SELECT \* FROM c WHERE c.type = ‘customer’ ORDER BY c.salesOrderCount DESC

Of course, we need to keep that salesOrderCount property in sync; each time we create a new sales order document, we also need to increment the salesOrderCount property in the related customer document. We could use the change feed like before, but stored procedures are a better choice when your updates are contained to a single logical partition.

In this case, the new sales order document is being written to the same logical partition as the related customer document. We can write a stored procedure in JavaScript that runs within the Azure Cosmos DB service which creates the new sales order document and updates the customer document with the incremented sales order count.

The big advantage here is that stored procedures in Azure Cosmos DB run as a transaction that succeeds or fails as a whole. Both write operations will need to complete successfully or they both roll back. This guarantees consistency between the salesOrderCount property in the customer document and the true number of related sales order documents in the same logical partition.

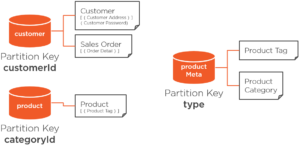
One last thing to mention is that this is a cross-partition query, unlike of our previous examples, which were all single-partition queries. Remember again that cross-partition queries aren’t necessarily evil, as long as they aren’t very common. In our case, this last query won’t run routinely on the website; it’s more like a “back office” query that an executive runs every now and again to find the top customers.

### Summary

This post has walked you through the steps to refactor a relational data model as non-relational for Azure Cosmos DB. We collapsed multiple entities by embedding, and we support denormalization through the use of the change feed and stored procedures.

We also combined customer and sales order documents in the same container, because they are both partitioned on the same value (customer ID). To wrap up the design, we can also combine the product category and product tag documents in a single “product metadata” container, since they are both partitioned on the same type property.

That brings us to our final design:

[](https://devblogs.microsoft.com/cosmosdb/wp-content/uploads/sites/52/2020/06/DM10-300x146.png)

Using a combination of non-relational modeling techniques, we’ve reduced nine tables into just three containers, where majority of queries run by the application are all scoped to a single logical partition.