**Azure DocumentDB vs. MongoDB**

A synthetic comparison

[DocumentDB](https://azure.microsoft.com/en-us/services/documentdb/) is a NoSQL *database-as-a-service* this is part of the Microsoft Azure platform. As a document store, it falls into the same category as MongoDB, CouchDB or RethinkDB and just like those, it handles documents in the JSON format.

When considering the integration of a NoSQL document store in their systems, many companies choose MongoDB because it’s among the most popular NoSQL engines out there and it has become very reliable over the past years. I feel that DocumentDB usually doesn’t get considered when making this decision although its characteristics make it a serious contender to MongoDB, even offering stronger advantages in some situations.

Because I feel that DocumentDB doesn’t get the love it deserves, I decided to write this synthetic and unbiased comparison between DocumentDB and MongoDB, supported by pointers to their respective documentations. I hope this can serve you as a guide when trying to weight the pros and cons of each platform.

**Updates to this post**

**[November 2016]** Removed all mentions of the lack of local emulator for DocumentDB as Microsoft announced the general availability of such a [local development version](https://docs.microsoft.com/en-us/azure/documentdb/documentdb-nosql-local-emulator). Note that the local emulator is only available for Windows currently (thanks [David Mason](https://medium.com/@nostriluu) for the suggested edit!).

**[November 2016]** Removed the mention of auto-expiring documents being a feature that is exclusive to DocumentDB, as [Bo Bendtsen](https://medium.com/@bobend) kindly pointed out that MongoDB has [similar capabilities](https://docs.mongodb.com/v3.2/core/index-ttl/).

**[January 2017]** Added a section about DocumentDB’s out-of-the-box, built-in security as suggested by [Mary Branscombe](https://medium.com/@marypcbuk).

**The similarities**

Conceptually, there are some basic similarities between the two databases:

* Documents are stored and served in the JSON format
* Documents can be retrieved using a rich query language that plays well with the JSON syntax

**Features unique to MongoDB**

Let’s start by enumerating the main MongoDB features that don’t have any (reasonably matching) DocumentDB counterpart.

**Rich query capabilities with the aggregation pipeline**

MongoDB’s [aggregation pipeline](https://docs.mongodb.com/manual/core/aggregation-pipeline/) is a very powerful feature that lets you build a pipeline composed of data processing stages, each filtering and transforming the documents coming from a collection. The possibilities offered by this pipeline are nearly limitless and its flexibility can cater for virtually any kind of query.

In comparison, DocumentDB’s SQL-like query syntax only allows simple filtering over the documents, even lacking “basic” constructs like *count* or *sum* (although they are [working on it](https://feedback.azure.com/forums/263030-documentdb/suggestions/6333963-add-support-for-aggregate-functions-like-count-su) and you can work around with server-side Javascript in the meantime). A handy query cheat sheet can be found [here](https://azure.microsoft.com/en-us/documentation/articles/documentdb-sql-query-cheat-sheet/).

**Map-reduce**

Somehow similar to the aggregation pipeline, MongoDB’s [map-reduce](https://docs.mongodb.com/manual/core/map-reduce/) feature lets a collection’s documents flow through 2 separate stages that iteratively transforms (or projects) then groups the documents. Both stages are defined in Javascript.

There isn’t any such concept in DocumentDB, although similar results can be achieved using stored procedures (see below).

**Full-text indexes**

Among the [different types of indexes](https://docs.mongodb.com/manual/indexes/) available on MongoDB, the *text* index offers [full-text search](https://docs.mongodb.com/manual/text-search/) capabilities.

DocumentDB doesn’t provide any full-text indexing. The recommended way to add full-text search to a DocumentDB database is to pair it with an Azure Search service; there is a good [integration story](https://azure.microsoft.com/en-us/documentation/articles/documentdb-search-indexer/) between the two.

**More platforms supported by client-side drivers**

I think it’s important to mention that MongoDB’s drivers support a very [large spectrum of platforms](https://docs.mongodb.com/ecosystem/drivers/), whereas DocumentDB only has SDKs for .NET, Java, Python and Node.js — but you can try your luck using any MongoDB driver with DocumentDB thanks to its [support for MongoDB’s protocol](https://azure.microsoft.com/en-us/documentation/articles/documentdb-protocol-mongodb/).

**Features unique to DocumentDB**

Now let’s do the inverse exercise and list the DocumentDB’s features that can’t be found in MongoDB.

**Server-side Javascript**

That’s a [key feature of DocumentDB](https://azure.microsoft.com/en-us/documentation/articles/documentdb-programming/). It has a rich server-side Javascript API, letting you create data processing functions. Those server-side functions can take 3 different forms:

* stored procedures that can do pretty much anything (inserting, querying, updating documents) and get called through the SDKs or the REST API
* triggers (or hooks) that get executed before or after specific operations (like on a document insertion for example)
* UDFs (user-defined functions) that can be called from and augment the SQL query language, somehow narrowing the gap with MongoDB’s rich query capabilities

Now MongoDB can execute server-side Javascript [as](https://docs.mongodb.com/manual/core/server-side-javascript/) [well](https://docs.mongodb.com/manual/tutorial/store-javascript-function-on-server/), but my understanding is that:

* Map-reduce and the *$where* query operator can only be used for queries, not updates
* The Javascript functions that you can store in the special *system* collection are only suitable for administration or maintenance purposes.

MongoDB’s documentation clearly states that there are performance limitations in executing server-side Javascript; in comparison, DocumentDB is really designed for this purpose as it pre-compiles your Javascript code, then stores and executes the resulting bytecode.

**Transactions**

Thanks to the Javascript stored procedures we’ve just mentioned, it is possible to run ACID transactions on a DocumentDB collection. The way it works is really simple: if your Javascript function completes, all write operations it has performed get committed; if the function throws any exception, all operations get rolled back.

There isn’t really any concept of transaction in MongoDB besides [single-document atomicity](https://docs.mongodb.com/manual/core/write-operations-atomicity/), which means that inserting or updating a document is guaranteed to be atomic, but a write operation involving multiple documents is not atomic as a whole.

**Full indexing by default**

DocumentDB takes a rather drastic approach to indexing: by default, it indexes all the fields of the documents you are storing! Many of you may see this as a waste of processing time and storage space — which honestly it is to some extent — but this gives the interesting advantage of offering excellent query performance out of the box. For those who prefer to have a better control over what gets indexed, it is always possible to define [custom indexing policies](https://azure.microsoft.com/en-us/documentation/articles/documentdb-indexing-policies/).

**(Easy) global distribution**

Another pretty recent addition to DocumentDB’s capabilities is [global distribution](https://azure.microsoft.com/en-us/documentation/articles/documentdb-distribute-data-globally/). Basically, this feature lets you scale your DocumentDB instance across different regions around the world and define what type of consistency you expect between the regions, from strong to eventual. It is even possible to configure an automatic and transparent failover over the different regions.

Of course, deploying a world-wide cluster of MongoDB nodes is certainly possible, but what I want to emphasize here is how easy it is to setup such a cluster. That’s obviously beyond DocumentDB’s core features and is related to its PaaS nature, but I don’t believe there is any service provider offering such geo-distributed setup for MongoDB (at that cost and ease of use).

**Security**

It’s worth mentioning that as a service, DocumentDB provides [built-in security and access control](https://docs.microsoft.com/en-us/azure/documentdb/documentdb-secure-access-to-data) that are there by default… No password-less admin access! And beyond that, it also gives the ability to control the access to collections and documents in a fine-grained fashion by creating users and linking them to those resources through password-protected permissions.

**Pricing**

The last, but certainly not least criteria of comparison to consider is the cost. But we should be careful not to compare apples and oranges here: DocumentDB belongs to the PaaS family whereas MongoDB is a database, not a service. So let’s take [mLab](https://mlab.com/), a MongoDB PaaS offering, as a point of comparison.

First I should clarify how DocumentDB is billed. Each collection is billed over 2 dimensions:

* storage used, at 0.25 USD per GB / month
* reserved Request Units per seconds, at ~6 USD per 100 RU / month

The number of RU you reserve dictates the guaranteed bandwidth you will get (want to learn more about Request Units? check [my post](https://medium.com/@th0maswe1ss/how-i-learned-to-stop-worrying-and-love-documentdbs-request-units-a05efef7df64)!). Basically, a RU represents “the processing required to read a single 1KB document with 10 properties”. That being said, it’s not easy to evaluate the actual cost of complex operations like big queries or elaborate stored procedure, although [this guide](https://azure.microsoft.com/en-us/documentation/articles/documentdb-request-units/) helps a lot. But we can do the reverse exercise of looking at how many RU we could get for the price of a mLab plan.

What I didn’t mention so far is that DocumentDB runs on local SSD, so in order to do a fair comparison, let’s take the “High Performance M3” plan from [this page](https://mlab.com/plans/pricing/), which at the time of this writing (September 2016) is priced at 1,390 USD monthly for 80GB of storage.

* Those 80GB would be billed 20 USD on DocumentDB
* That leaves 1,370 USD, or more than 22,800 RU

I mentioned before that it’s difficult to evaluate the “value” of a RU, but from my experience, 22,800 is *a lot*, something in the range of 200 complex queries per second. And even though it’s similarly difficult to evaluate the capabilities of that “High Performance M3” plan, I would say that we are playing on a similar scale, or at least not orders of magnitude different.

Besides, what’s nice with the elasticity of RU is that it is designed to be a unit of scale, which means that you can start with a modest amount of RU and (seamlessly) scale it out as the usage of your collections increases, while still taking advantage of local SSD performance from the beginning.

**What about the cost of vendor lock-in?**

A concern expressed by many is vendor lock-in: if I use DocumentDB, I’m not only locked with Microsoft, but also with Azure as a platform. You could even argue that the lack of such lock-in should have been listed in MongoDB’s advantages over DocumentDB. I agree. But what’s the real cost of this lock-in?

DocumentDB stores documents in the JSON format. That’s a standard format used by most NoSQL databases (hey, even [SQL Server speaks JSON](https://msdn.microsoft.com/en-us/library/dn921897.aspx)!), so moving your documents out of DocumentDB and injecting them in some other database should not be an issue.

The main technical lock-in you have to deal with is the query interface: each database has its own way of querying documents. Most of the time, you perform those queries through some SDK or driver, so from the perspective of your application code, the lock-in or adherence to a particular database comes mainly from the interface of that SDK. But then, if your developers are doing it right, that interface should be encapsulated behind some kind of data access interface that hides the implementation details to the rest of the application.

And if your concern is that you may want to migrate to MongoDB at a later stage, remember that DocumentDB has protocol-compatibility with MongoDB, which means that you can use any MongoDB driver to access DocumentDB and perform [most of the CRUD operations](https://azure.microsoft.com/en-us/documentation/articles/documentdb-mongodb-guidelines/).

**How I recommend to guide your decision**

Wrapping it up, here are the first questions I think you could ask yourself when having to make a choice between those databases (in no particular order):

* **Complexity of queries**: do your queries require the full power of MongoDB’s aggregation pipeline, or can you implement them with DocumentDB’s SQL and some server-side Javascript?
* **Transactions**: does your business logic require collection-wide, multi-document transactions, or is MongoDB’s single-document atomicity enough for your requirements?

Based on your answers and the general direction they give, you can then refine your analysis and consider the rest of the features I mentioned (full-text search, global distribution etc.)

**Please comment!**

I tried to perform this comparison in the most honest and unbiased way, but I could be wrong on some aspects. So feel free to reach out if you feel that some features are missing, or were over- or underestimated! I intend this post to evolve over time and get complemented to become a good reference on the comparison between DocumentDB and MongoDB.

Thanks for reading!