**David Chappell**

# Introducing DocumentDB

## A NoSQL Database for Microsoft Azure

**Sponsored by Microsoft Corporation**

## Contents

## Why DocumentDB?

Suppose you're responsible for creating a new application. You're not entirely sure what kinds of data it will work with, although you know that it will be used by a variety of clients. You're also not sure how that data should be structured; things are bound to change over the application's lifetime. You're not even sure how much data the application will need to handle.

You do know some things, however. You know you want the application to run in the public cloud, for all of the usual reasons: fast deployment, low cost, scalability, and more. You know that the application needs to be available all the time, which means that downtime for things like schema changes isn't an option. You know that you'll need powerful queries and atomic transactions—simple reads and writes aren't enough. And you know that you'd like to build on your existing knowledge rather than be forced to grapple with an entirely unfamiliar technology.

You could use a relational database for this application. On Microsoft Azure, for example, you might use SQL Database, which is a managed database service, or you might run your own database server in a virtual machine. But going this route requires defining a schema up front, then probably accepting downtime whenever you modify that schema to handle changes in the structure of your data. Relational databases can also be hard to scale for lots of users, and using one means addressing the challenge of object/relational mapping.

Is there another option? There is; instead of a relational database, your application can use DocumentDB, a managed NoSQL database provided by Microsoft Azure.

DocumentDB is designed for situations like the one just described. It doesn't require any kind of schema, opting instead to store data as JavaScript Object Notation (JSON). This frees application developers from being locked into a hard-to-change structure for their data. It also frees them from worrying about object/relational mapping, since the state of their application's objects can typically be stored directly as JSON. And because DocumentDB is a managed Azure service, a developer can create a new database in minutes, then let DocumentDB handle much of the management. All of this makes development, deployment, and updating simpler and faster.

To support applications with lots of users and lots of data, DocumentDB is designed to scale: a single database can be spread across many different machines, and it can contain hundreds of terabytes of data. DocumentDB also provides a query language based on SQL, along with the ability to run JavaScript code directly in the database as stored procedures and triggers with atomic transactions.

The truth is that applications are different today, and the way they work with data is different, too. Database technologies are evolving to reflect these changes, as DocumentDB shows. And this NoSQL database service isn't difficult to understand—you just need to grasp a few fundamental concepts. Those concepts include:

- The DocumentDB data model.
- How applications work with data.
- The options applications have for balancing performance with data consistency.

What follows looks at each of these.

## The DocumentDB Data Model

DocumentDB's data model is simple: all data is stored in JSON *documents*[1]. For example, suppose you're creating a DocumentDB application that works with customers. Information about each of those customers would typically be described in its own JSON document, and so the document for the customer Contoso might look like this:

```
{
  "name": "Contoso",
  "country": "Germany",
  "contacts":
   [
     {"admin": "Johann Schmidt", "email": "johschmidt@contoso.com"},
     {"purchasing": "Anusha Swarmi", "email": "anusha@contoso.com"}
   ],
  "salesYTD": 49003.23
}
```

The document for the customer Fabrikam would likely be similar, but it needn't be identical. It might look like this:

```
{
  "name": "Fabrikam",
  "country": "USA",
  "contacts":
   [
     {"ceo": "Mary Chen", "email": "mary@fabrikam.com",
      "phone": "510-555-3443"},
     {"purchasing": "Frank Allen", "email": "franka@fabrikam.com",
      "email": "fallen@fabrikam.com"}
   ],
  "salesRank": 3,
  "salesYTD": 1399450.22
}
```

As these simple documents show, JSON data is modelled as name/value pairs. In this example, each customer has elements for name and country, with an appropriate value for each one. Each also has a contacts element, the value for which is an array of name/value pairs wrapped in square brackets. An element's values can be character strings, integers, floating point numbers, or another JSON type.

Notice that while these two customer documents are similar, their structure isn't identical. This is fine; DocumentDB doesn't enforce any schema. In this example, both customers have several common elements, such as name and country. There are also differences, however. The contacts for Fabrikam include the CEO—they're a big customer—along with her phone number. Also, the purchasing manager for Fabrikam has two email addresses, rather than the single address for Contoso's purchasing manager, and the Fabrikam document contains an element describing its sales rank. This is all perfectly legal in DocumentDB. Since there's no schema, there's no requirement to make all documents conform to the same structure.

[1] As its name suggests, DocumentDB fits in the NoSQL category known as *document databases*. It's not a key/value store like Azure Tables or a column family store like HBase.

DocumentDB groups JSON documents into *collections*. A single DocumentDB database can contain many collections; to grow the database, you just add a new collection. Figure 1 shows how this looks.

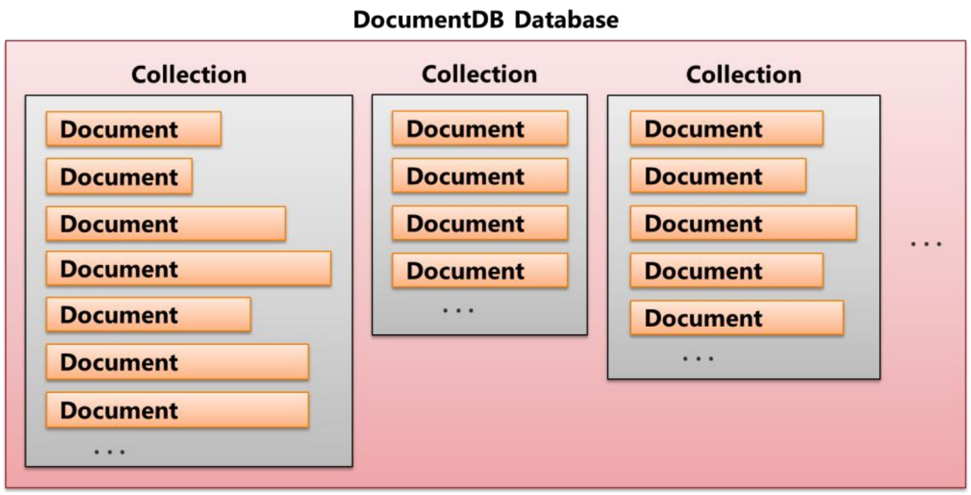


**Figure 1: A DocumentDB database contains collections of JSON documents.**

As the figure suggests, the documents in a particular collection might all look quite similar, with each one containing, say, the information for a specific customer in the style shown earlier. It's also possible for each document in a collection to have a completely different structure—DocumentDB doesn't constrain this. Unlike a relational table, where every row holds data in a fixed set of columns, a document can contain whatever the application needs. And although it's not shown in Figure 1, documents can have attachments such as videos that are accessible via DocumentDB but are physically stored in Azure Blobs or elsewhere.

With DocumentDB, an application typically keeps all of the data about some entity, such as a customer, in a single document. Unlike a relational database, which would probably spread that data across several different tables, applications using DocumentDB commonly keep it all together. While the style used by a relational database has some advantages, storing all of an object's data in one place can make life simpler for application developers. Rather than accessing their data using complex queries with one or more joins, for example, they can instead work directly with a document containing everything they need. This approach also speeds up access, since a DocumentDB request can often look at just one document to find what's needed.

## Working with Data

DocumentDB clients can be written in multiple languages, including C#, JavaScript, and Python. Whatever choice a developer makes, the client accesses DocumentDB through RESTful access methods. A developer can use these to work with documents in a collection in a few different ways. The options are:

- Using these access methods directly for create/read/update/delete (CRUD) operations.
- Submitting requests expressed in the DocumentDB query language.

- Defining and executing logic that runs inside DocumentDB, including stored procedures, triggers, and user-defined functions (UDFs).

Figure 2 illustrates these options.

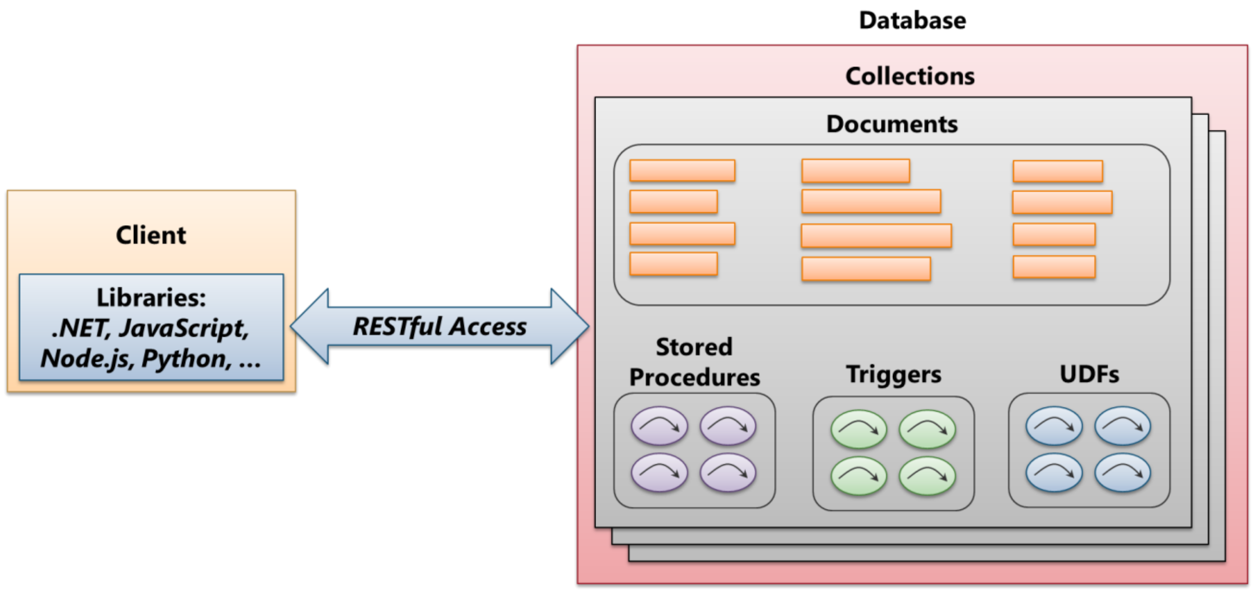


**Figure 2: Clients access documents in collections via RESTful access methods and can also run logic in the database itself.**

## RESTful Access Methods

If an application has the necessary permissions, it can use DocumentDB's RESTful access methods to perform CRUD operations on documents and other resources. Like every RESTful interface, DocumentDB uses the standard HTTP verbs:

- A GET request returns the value of a resource, such as a document.
- A PUT request replaces a resource.
- A POST request creates a new resource. POSTs are also used to send requests using the DocumentDB query language and to create new stored procedures, triggers, and UDFs.
- A DELETE request removes a resource.

A developer using this interface is free to construct requests manually—it's just REST. But to make life easier, DocumentDB provides several client libraries. As Figure 2 shows, the options include .NET (with LINQ support), JavaScript, Node.js, and Python.

***DocumentDB's Native Tongue: JavaScript***

Whether an application uses DocumentDB's client libraries or directly invokes its RESTful access methods, the code can be written in many different languages. Still, it's fair to say that the native tongue of DocumentDB is JavaScript.

One reason for this is that DocumentDB returns results in JSON. A JavaScript application can use a JSON parser to turn these results directly into JavaScript variables, modify those variables, then send the changed data back to the database. This is a simple, natural way to work; there's no impedance mismatch, which means there's also no need for mapping, object/relational or otherwise.

Also, logic that executes within DocumentDB itself, including stored procedures, triggers, and UDFs, must be written in JavaScript, as described later. While developers working in C# or other languages can certainly use DocumentDB successfully, writing code that runs in the database will require learning JavaScript or finding somebody who knows it. Given the large (and growing) number of developers who already work in this language, it shouldn't be surprising that DocumentDB's creators chose to make it a fundamental part of this cloud database service.

## The DocumentDB Query Language

A DocumentDB client can read and write data using the service's RESTful access methods. But a real database needs a real query language, something that lets applications work with data in more complex ways. This is what the DocumentDB query language provides.

This language is an extended subset of SQL, a technology that many developers already know. For example, suppose the simple JSON documents shown earlier are contained in a collection called *customers*. Here's a query on that collection:

```
SELECT c.salesYTD
FROM customers c
WHERE c.name = "Fabrikam"
```

As anybody who knows SQL can probably figure out, `SELECT` requests the value of the element salesYTD, `FROM` indicates that the query should be executed against documents in the customers collection, and `WHERE` specifies the condition that documents within that collection should meet. The query's result is year-to-date sales for Fabrikam formatted as JSON data:

```
{
    "salesYTD": 1399450.22
}
```

### *Indexing in DocumentDB*

Indexes are an important aspect of database technologies. Creating an index makes lookups faster, and so operations on indexed elements will have better performance. Some NoSQL databases, such as many key/value stores, provide just a single index. Others, such as relational databases and some document databases, let their users explicitly create indexes on particular elements.

DocumentDB takes neither of these paths. Instead, it by default creates an index on every JSON element in every document in a collection. In the example documents shown earlier, for example, DocumentDB would automatically create indexes on name, country, contacts, and more. Developers don't need to decide up front which JSON elements they're likely to query on, then create indexes only for those elements. DocumentDB automatically indexes all of them (and advanced users can configure and tune these indexes as needed). This gives ad hoc queries speedy access to everything in the database.

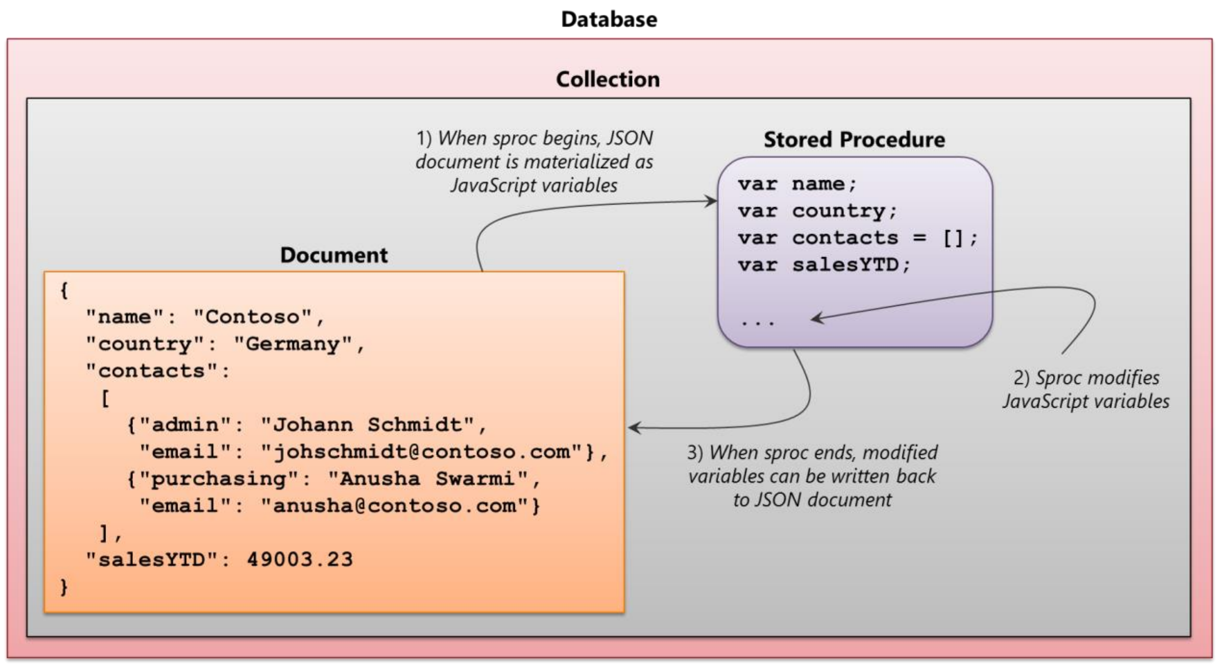
## Executing Logic in the Database

The DocumentDB query language lets a client issue a request that's parsed and executed when it's received. But there are plenty of situations where it makes more sense to run logic stored in the database itself. DocumentDB provides several ways to do this, including stored procedures (commonly called *sprocs*), triggers, and user-defined functions (UDFs). A collection can contain any or all of them.

### Stored Procedures

Stored procedures implement logic, which means they must be written in some programming language. Relational databases commonly create their own language for doing this, such as SQL Server's T-SQL. But what should this language look like for a database that stores JSON documents? The answer is obvious: stored procedures should be written in JavaScript, which is exactly what DocumentDB does.

To execute a stored procedure, a client application issues a POST request indicating which sproc to run and passing in any input parameters. Figure 2 illustrates how the sproc works with documents in a collection.


Database
Collection
1) When sproc begins, JSON document is materialized as JavaScript variables
Stored Procedure
Document
2) Sproc modifies JavaScript variables
3) When sproc ends, modified variables can be written back to JSON document

```
var name;
var country;
var contacts = [];
var salesYTD;

...
```

```
{
  "name": "Contoso",
  "country": "Germany",
  "contacts":
   [
     {"admin": "Johann Schmidt",
      "email": "johschmidt@contoso.com"},
     {"purchasing": "Anusha Swarmi",
      "email": "anusha@contoso.com"}
   ],
  "salesYTD": 49003.23
}
```

**Figure 3: A stored procedure is JavaScript code that works directly with document elements as variables.**

As the figure shows, sprocs work with documents in a straightforward way. When the sproc begins, elements of the JSON document (or documents) it's working with are copied into JavaScript variables (step 1). The sproc's code then works with those variables, changed them as needed (step 2). When the sproc completes, any modified variables can have their values written back to the JSON document (step 3). The goal is to make writing sprocs as simple and natural as possible—they're just JavaScript.

Every sproc is wrapped in an atomic transaction. If the sproc ends normally, all of the changes it has made to documents in this collection will be committed. If it throws an exception, however, all of the changes it has made to these documents will be rolled back. And while the sproc is executing, its work is isolated—no other requests to this database will see partial results.

Stored procedures can make life easier for developers, since logic that might otherwise be replicated in multiple applications can instead be encapsulated in the database. Stored procedures can also have performance advantages. Rather than requiring an application to issue multiple requests to accomplish a task, for example, with the round trips this implies, a sproc can do all of this work with a single call. While stored procedures aren't right for every situation, they're definitely a useful and important part of modern databases.

## Triggers

DocumentDB triggers are similar in some ways to stored procedures: they're invoked via a POST request, and they're written in JavaScript. They also materialize JSON documents into JavaScript variables and are automatically wrapped in an atomic transaction. Unlike sprocs, however, a trigger runs when a specific event happens, such as data being created, changed, or deleted.

DocumentDB supports pre-triggers, which run before the event occurs, and post-triggers, which run after the event has finished. For example, a pre-trigger executed when a document is changed might do data validation, making sure that the new data conforms to a specific format. A post-trigger run when a document is created might update another document in the collection that tracks all newly created information. If an application creates and registers these two triggers, future requests to change or add documents can indicate that the database should also run the appropriate trigger. Rather than requiring the application to explicitly perform data validation and document tracking itself, it can rely on the triggers to handle these.

If a trigger throws an exception, the transaction it's part of aborts, and everything gets rolled back. This includes the work done by the trigger itself and the work done by whatever request caused the trigger to execute. For example, if a post-trigger run on document creation aborts, the new document will not be created.

Triggers are a useful way to carry out common database functions, and like stored procedures, they're an integral part of modern databases.

### User-Defined Functions

Like stored procedures and triggers, user-defined functions are written in JavaScript, and they run within DocumentDB itself. UDFs can't make changes to the database, however—they're read-only. Instead, a UDF provides a way to extend the DocumentDB query language with custom code.

For example, suppose the customers collection contained a UDF called calculateTax that computed the tax on sales. To find all customers where the tax is more than $1,000, an application might issue a query like this:

```
SELECT *
FROM customers c
WHERE calculateTax(c.salesYTD)> 1000
```

Putting this calculation in a UDF makes it easier to use, since it acts like part of the query language. It also makes the logic simpler to share, since it's stored in the database rather than in a single application.

UDFs can do quite a bit more than this. Since they're written in JavaScript, they make it straightforward to add standard JavaScript functions to the DocumentDB query language. They can also be used to check for the presence of elements in a document, such as returning all customer documents that have a salesRank element, or implementing geospatial queries, such as NEAR, or many other things. The ability to extend the query language with custom JavaScript code can make life significantly easier for the people who use DocumentDB.

### *Managing Resources*

DocumentDB is a multi-tenant service. In other words, many different client applications use it at the same time. But suppose one of those clients runs a stored procedure or a trigger or a UDF that eats up a huge share of the service's resources. Won't this cause performance to suffer for the other clients?

Fortunately, this can't happen with DocumentDB. The reason is that a DocumentDB user pays for a specific number of capacity units (CUs), each of which provides a defined amount of storage and throughput. If you need more of either, you can pay for more CUs. If your requirements decrease, you can use fewer CUs.

A stored procedure or other request that exceeds the number of CUs you've paid for will be stopped—it can't use resources that are reserved for other users. The result is that applications are guaranteed a specific performance level for database requests regardless of who else might be using the service at the same time.

## Consistency Options

DocumentDB is designed to be both scalable and reliable. To achieve this, it maintains at least three copies of all data, storing each copy on a different physical server. This replication is done at the granularity of collections: a single server can store multiple collections, but a collection is never split across servers.

Replication helps scalability because different clients reading data in the same collection can potentially have their requests handled by any of the replicas—a single server won't be a bottleneck. Replicating each collection also helps reliability by ensuring that data is still available even if one or two servers become inaccessible.

But replication isn't free. The biggest challenge it brings happens when database clients write data. How can a document be changed while still keeping all three replicas consistent? DocumentDB handles this by designating one replica as the primary and the others as secondaries. All writes to a collection are made first to the primary replica, then propagated to the secondaries.

Still, propagating a write from the primary to the secondaries takes some time. While it's happening, either the same application or another one might read this just-modified data. If this read is handled by the primary or by a secondary replica that's already been updated, life is good; the application will see the correct (i.e. the newest) data. But suppose the read is handled by one of the collection's secondaries that hasn't yet been informed of the latest write. In this case, the read will return out-of-date data.

What's the best way to handle this situation? The answer depends on the application. In some cases, applications absolutely need to see the most current data on every read. But some applications can accept reading slightly out-of-date information. Doing this improves the application's performance and availability, since it can read from replicas other than the primary.

Because different applications have different requirements, DocumentDB doesn't mandate a choice. Instead, it defines four distinct consistency options, each with different tradeoffs between data correctness and performance. The choices are:

- *Strong:* A DocumentDB client always sees completely consistent data. The tradeoff is that reads and writes are slower than with the other three options. An application that must always see the most current data, such as banking software that moves money between documents, might choose this option.
- *Session:* A client will always read its own writes, but other clients reading this same data might see older values. An application that works on behalf of a specific user, such as a blogging application, might choose this option. In cases like these, each user expects to see the changes she makes, i.e., everything done in her session, right away. Yet she probably doesn't care whether there's a slight delay in seeing changes made by other users. This turns out to be the sweet spot for many applications—it has the best trade-off between correctness and performance—and so it's the default in DocumentDB.
- *Bounded Staleness:* A client might see old data, but it can specify a limit for how old that data can be, e.g., one second. This is similar to Session consistency, but it lets reads be a little faster while still preserving the order of updates. In a multi-player game, for instance, which requires great performance and strict ordering of events, Bounded Staleness might be the right choice.
- *Eventual:* A client might see old data for as long as it takes a write to propagate to all replicas. This option has the highest performance, but a client might sometimes read out-of-date information or see updates out of order.

Even though all DocumentDB databases default to Session consistency, developers are free to choose a weaker option for each read their application makes. It's also possible to change the default if necessary. But because different applications really do have different requirements, DocumentDB doesn't make the choice for them. Developers are free to use the consistency option that's best for their situation.

## Conclusion

DocumentDB is a relatively simple and scalable database—it's a NoSQL technology—that also provides more advanced data management capabilities such as a SQL-based query language, stored procedures, and atomic transactions. You should consider using it whenever your application needs any or all of the following:

- The programming ease provided by native JSON and JavaScript support.
- The flexibility of not being locked into a schema.
- The scale and availability allowed by replicating data across multiple machines.
- The simplicity of a managed database service on a public cloud platform.

As computing continues its move to the cloud, more and more applications can benefit from this approach. In fact, a cloud platform that doesn't offer a document database today is probably behind the times.

## About the Author

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technologies.