# SQL query and SQL syntax in DocumentDB

Microsoft Azure DocumentDB supports querying documents using SQL (Structured Query Language) as a JSON query language. DocumentDB is truly schema-free. By virtue of its commitment to the JSON data model directly within the database engine, it provides automatic indexing of JSON documents without requiring explicit schema or creation of secondary indexes.

While designing the query language for DocumentDB we had two goals in mind:

* Instead of inventing a new JSON query language, we wanted to support SQL. SQL is one of the most familiar and popular query languages. DocumentDB SQL provides a formal programming model for rich queries over JSON documents.
* As a JSON document database capable of executing JavaScript directly in the database engine, we wanted to use JavaScript’s programming model as the foundation for our query language. The DocumentDB SQL is rooted in JavaScript’s type system, expression evaluation, and function invocation. This in-turn provides a natural programming model for relational projections, hierarchical navigation across JSON documents, self joins, spatial queries, and invocation of user defined functions (UDFs) written entirely in JavaScript, among other features.

We believe that these capabilities are key to reducing the friction between the application and the database and are crucial for developer productivity.

We recommend getting started by watching the following video, where Aravind Ramachandran shows DocumentDB’s querying capabilities, and by visiting our [Query Playground](http://www.documentdb.com/sql/demo), where you can try out DocumentDB and run SQL queries against our dataset.

[AZURE.VIDEO dataexposedqueryingdocumentdb]

Then, return to this article, where we’ll start with a SQL query tutorial that walks you through some simple JSON documents and SQL commands.

## Getting started with SQL commands in DocumentDB

To see DocumentDB SQL at work, let’s begin with a few simple JSON documents and walk through some simple queries against it. Consider these two JSON documents about two families. Note that with DocumentDB, we do not need to create any schemas or secondary indices explicitly. We simply need to insert the JSON documents to a DocumentDB collection and subsequently query. Here we have a simple JSON document for the Andersen family, the parents, children (and their pets), address and registration information. The document has strings, numbers, booleans, arrays and nested properties.

**Document**

{  
 "id": "AndersenFamily",  
 "lastName": "Andersen",  
 "parents": [  
 { "firstName": "Thomas" },  
 { "firstName": "Mary Kay"}  
 ],  
 "children": [  
 {  
 "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 }  
 ],  
 "address": { "state": "WA", "county": "King", "city": "seattle" },  
 "creationDate": 1431620472,  
 "isRegistered": true  
}

Here’s a second document with one subtle difference – givenName and familyName are used instead of firstName and lastName.

**Document**

{  
 "id": "WakefieldFamily",  
 "parents": [  
 { "familyName": "Wakefield", "givenName": "Robin" },  
 { "familyName": "Miller", "givenName": "Ben" }  
 ],  
 "children": [  
 {  
 "familyName": "Merriam",   
 "givenName": "Jesse",   
 "gender": "female", "grade": 1,  
 "pets": [  
 { "givenName": "Goofy" },  
 { "givenName": "Shadow" }  
 ]  
 },  
 {   
 "familyName": "Miller",   
 "givenName": "Lisa",   
 "gender": "female",   
 "grade": 8 }  
 ],  
 "address": { "state": "NY", "county": "Manhattan", "city": "NY" },  
 "creationDate": 1431620462,  
 "isRegistered": false  
}

Now let’s try a few queries against this data to understand some of the key aspects of DocumentDB SQL. For example, the following query will return the documents where the id field matches AndersenFamily. Since it’s a SELECT \*, the output of the query is the complete JSON document:

**Query**

SELECT \*   
FROM Families f   
WHERE f.id = "AndersenFamily"

**Results**

[{  
 "id": "AndersenFamily",  
 "lastName": "Andersen",  
 "parents": [  
 { "firstName": "Thomas" },  
 { "firstName": "Mary Kay"}  
 ],  
 "children": [  
 {  
 "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 }  
 ],  
 "address": { "state": "WA", "county": "King", "city": "seattle" },  
 "creationDate": 1431620472,  
 "isRegistered": true  
}]

Now consider the case where we need to reformat the JSON output in a different shape. This query projects a new JSON object with two selected fields, Name and City, when the address’ city has the same name as the state. In this case, “NY, NY” matches.

**Query**

SELECT {"Name":f.id, "City":f.address.city} AS Family   
FROM Families f   
WHERE f.address.city = f.address.state

**Results**

[{  
 "Family": {  
 "Name": "WakefieldFamily",   
 "City": "NY"  
 }  
}]

The next query returns all the given names of children in the family whose id matches WakefieldFamily ordered by the city of residence.

**Query**

SELECT c.givenName   
FROM Families f   
JOIN c IN f.children   
WHERE f.id = 'WakefieldFamily'  
ORDER BY f.address.city ASC

**Results**

[  
 { "givenName": "Jesse" },   
 { "givenName": "Lisa"}  
]

We would like to draw attention to a few noteworthy aspects of the DocumentDB query language through the examples we’ve seen so far:

* Since DocumentDB SQL works on JSON values, it deals with tree shaped entities instead of rows and columns. Therefore, the language lets you refer to nodes of the tree at any arbitrary depth, like Node1.Node2.Node3…..Nodem, similar to relational SQL referring to the two part reference of <table>.<column>.
* The structured query language works with schema-less data. Therefore, the type system needs to be bound dynamically. The same expression could yield different types on different documents. The result of a query is a valid JSON value, but is not guaranteed to be of a fixed schema.
* DocumentDB only supports strict JSON documents. This means the type system and expressions are restricted to deal only with JSON types. Please refer to the [JSON specification](http://www.json.org/) for more details.
* A DocumentDB collection is a schema-free container of JSON documents. The relations in data entities within and across documents in a collection are implicitly captured by containment and not by primary key and foreign key relations. This is an important aspect worth pointing out in light of the intra-document joins discussed later in this article.

## DocumentDB indexing

Before we get into the DocumentDB SQL syntax, it is worth exploring the indexing design in DocumentDB.

The purpose of database indexes is to serve queries in their various forms and shapes with minimum resource consumption (like CPU and input/output) while providing good throughput and low latency. Often, the choice of the right index for querying a database requires much planning and experimentation. This approach poses a challenge for schema-less databases where the data doesn’t conform to a strict schema and evolves rapidly.

Therefore, when we designed the DocumentDB indexing subsystem, we set the following goals:

* Index documents without requiring schema: The indexing subsystem does not require any schema information or make any assumptions about schema of the documents.
* Support for efficient, rich hierarchical, and relational queries: The index supports the DocumentDB query language efficiently, including support for hierarchical and relational projections.
* Support for consistent queries in face of a sustained volume of writes: For high write throughput workloads with consistent queries, the index is updated incrementally, efficiently, and online in the face of a sustained volume of writes. The consistent index update is crucial to serve the queries at the consistency level in which the user configured the document service.
* Support for multi-tenancy: Given the reservation based model for resource governance across tenants, index updates are performed within the budget of system resources (CPU, memory, and input/output operations per second) allocated per replica.
* Storage efficiency: For cost effectiveness, the on-disk storage overhead of the index is bounded and predictable. This is crucial because DocumentDB allows the developer to make cost based tradeoffs between index overhead in relation to the query performance.

Refer to the [DocumentDB samples](https://github.com/Azure/azure-documentdb-net) on MSDN for samples showing how to configure the indexing policy for a collection. Let’s now get into the details of the DocumentDB SQL syntax.

## Basics of a DocumentDB SQL query

Every query consists of a SELECT clause and optional FROM and WHERE clauses per ANSI-SQL standards. Typically, for each query, the source in the FROM clause is enumerated. Then the filter in the WHERE clause is applied on the source to retrieve a subset of JSON documents. Finally, the SELECT clause is used to project the requested JSON values in the select list.

SELECT [TOP <top\_expression>] <select\_list>   
[FROM <from\_specification>]   
[WHERE <filter\_condition>]  
[ORDER BY <sort\_specification]

## FROM clause

The FROM <from\_specification> clause is optional unless the source is filtered or projected later in the query. The purpose of this clause is to specify the data source upon which the query must operate. Commonly the whole collection is the source, but one can specify a subset of the collection instead.

A query like SELECT \* FROM Families indicates that the entire Families collection is the source over which to enumerate. A special identifier ROOT can be used to represent the collection instead of using the collection name. The following list contains the rules that are enforced per query:

* The collection can be aliased, such as SELECT f.id FROM Families AS f or simply SELECT f.id FROM Families f. Here f is the equivalent of Families. AS is an optional keyword to alias the identifier.
* Note that once aliased, the original source cannot be bound. For example, SELECT Families.id FROM Families f is syntactically invalid since the identifier “Families” cannot be resolved anymore.
* All properties that need to be referenced must be fully qualified. In the absence of strict schema adherence, this is enforced to avoid any ambiguous bindings. Therefore, SELECT id FROM Families f is syntactically invalid since the property id is not bound.

### Sub-documents

The source can also be reduced to a smaller subset. For instance, to enumerating only a sub-tree in each document, the sub-root could then become the source, as shown in the following example.

**Query**

SELECT \*   
FROM Families.children

**Results**

[  
 [  
 {  
 "firstName": "Henriette Thaulow",  
 "gender": "female",  
 "grade": 5,  
 "pets": [  
 {  
 "givenName": "Fluffy"  
 }  
 ]  
 }  
 ],  
 [  
 {  
 "familyName": "Merriam",  
 "givenName": "Jesse",  
 "gender": "female",  
 "grade": 1  
 },  
 {  
 "familyName": "Miller",  
 "givenName": "Lisa",  
 "gender": "female",  
 "grade": 8  
 }  
 ]  
]

While the above example used an array as the source, an object could also be used as the source, which is what’s shown in the following example. Any valid JSON value (not undefined) that can be found in the source will be considered for inclusion in the result of the query. If some families don’t have an address.state value, they will be excluded in the query result.

**Query**

SELECT \*   
FROM Families.address.state

**Results**

[  
 "WA",   
 "NY"  
]

## WHERE clause

The WHERE clause (**WHERE <filter\_condition>**) is optional. It specifies the condition(s) that the JSON documents provided by the source must satisfy in order to be included as part of the result. Any JSON document must evaluate the specified conditions to “true” to be considered for the result. The WHERE clause is used by the index layer in order to determine the absolute smallest subset of source documents that can be part of the result.

The following query requests documents that contain a name property whose value is AndersenFamily. Any other document that does not have a name property, or where the value does not match AndersenFamily is excluded.

**Query**

SELECT f.address  
FROM Families f   
WHERE f.id = "AndersenFamily"

**Results**

[{  
 "address": {  
 "state": "WA",   
 "county": "King",   
 "city": "seattle"  
 }  
}]

The previous example showed a simple equality query. DocumentDB SQL also supports a variety of scalar expressions. The most commonly used are binary and unary expressions. Property references from the source JSON object are also valid expressions.

The following binary operators are currently supported and can be used in queries as shown in the following examples:

Arithmetic

+,-,\*,/,%

Bitwise

|, &, ^, <<, >>, >>> (zero-fill right shift)

Logical

AND, OR, NOT

Comparison

=, !=, <, >, <=, >=, <>

String

|| (concatenate)

Let’s take a look at some queries using binary operators.

SELECT \*   
FROM Families.children[0] c  
WHERE c.grade % 2 = 1 -- matching grades == 5, 1  
  
SELECT \*   
FROM Families.children[0] c  
WHERE c.grade ^ 4 = 1 -- matching grades == 5  
  
SELECT \*  
FROM Families.children[0] c  
WHERE c.grade >= 5 -- matching grades == 5

The unary operators +,-, ~ and NOT are also supported, and can be used inside queries as shown in the following example:

SELECT \*  
FROM Families.children[0] c  
WHERE NOT(c.grade = 5) -- matching grades == 1  
  
SELECT \*  
FROM Families.children[0] c  
WHERE (-c.grade = -5) -- matching grades == 5

In addition to binary and unary operators, property references are also allowed. For example, SELECT \* FROM Families f WHERE f.isRegistered returns the JSON document containing the property isRegistered where the property’s value is equal to the JSON true value. Any other values (false, null, Undefined, <number>, <string>, <object>, <array>, etc.) leads to the source document being excluded from the result.

### Equality and comparison operators

The following table shows the result of equality comparisons in DocumentDB SQL between any two JSON types.

<tr>  
 <td valign="top">  
 <strong>Op</strong>  
 </td>  
 <td valign="top">  
 <strong>Undefined</strong>  
 </td>  
 <td valign="top">  
 <strong>Null</strong>  
 </td>  
 <td valign="top">  
 <strong>Boolean</strong>  
 </td>  
 <td valign="top">  
 <strong>Number</strong>  
 </td>  
 <td valign="top">  
 <strong>String</strong>  
 </td>  
 <td valign="top">  
 <strong>Object</strong>  
 </td>  
 <td valign="top">  
 <strong>Array</strong>  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Undefined<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Null<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>OK</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Boolean<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>OK</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Number<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>OK</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>String<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>OK</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Object<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>OK</strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 </tr>  
 <tr>  
 <td valign="top">  
 <strong>Array<strong>  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 Undefined  
 </td>  
 <td valign="top">  
 <strong>OK</strong>  
 </td>  
 </tr>

For other comparison operators such as >, >=, !=, < and <=, the following rules apply:

* Comparison across types results in Undefined.
* Comparison between two objects or two arrays results in Undefined.

If the result of the scalar expression in the filter is Undefined, the corresponding document would not be included in the result, since Undefined doesn’t logically equate to “true”.

### BETWEEN keyword

You can also use the BETWEEN keyword to express queries against ranges of values like in ANSI SQL. BETWEEN can be used against strings or numbers.

For example, this query returns all family documents in which the first child’s grade is between 1-5 (both inclusive).

SELECT \*  
FROM Families.children[0] c  
WHERE c.grade BETWEEN 1 AND 5

Unlike in ANSI-SQL, you can also use the BETWEEN clause in the FROM clause like in the following example.

SELECT (c.grade BETWEEN 0 AND 10)  
FROM Families.children[0] c

For faster query execution times, remember to create an indexing policy that uses a range index type against any numeric properties/paths that are filtered in the BETWEEN clause.

The main difference between using BETWEEN in DocumentDB and ANSI SQL is that you can express range queries against properties of mixed types – for example, you might have “grade” be a number (5) in some documents and strings in others (“grade4”). In these cases, like in JavaScript, a comparison between two different types results in “undefined”, and the document will be skipped.

### Logical (AND, OR and NOT) operators

Logical operators operate on Boolean values. The logical truth tables for these operators are shown in the following tables.

|  |  |  |  |
| --- | --- | --- | --- |
| OR | True | False | Undefined |
| True | True | True | True |
| False | True | False | Undefined |
| Undefined | True | Undefined | Undefined |

|  |  |  |  |
| --- | --- | --- | --- |
| AND | True | False | Undefined |
| True | True | False | Undefined |
| False | False | False | False |
| Undefined | Undefined | False | Undefined |

|  |  |
| --- | --- |
| NOT |  |
| True | False |
| False | True |
| Undefined | Undefined |

### IN keyword

The IN keyword can be used to check whether a specified value matches any value in a list. For example, this query returns all family documents where the id is one of “WakefieldFamily” or “AndersenFamily”.

SELECT \*  
FROM Families   
WHERE Families.id IN ('AndersenFamily', 'WakefieldFamily')

This example returns all documents where the state is any of the specified values.

SELECT \*  
FROM Families   
WHERE Families.address.state IN ("NY", "WA", "CA", "PA", "OH", "OR", "MI", "WI", "MN", "FL")

IN is equivalent to chaining multiple OR clauses, however since it can be served using a single index, DocumentDB supports a higher [limit](/documentation/articles/documentdb-limits) for the number of arguments specified within an IN clause.

### Ternary (?) and Coalesce (??) operators

The Ternary and Coalesce operators can be used to build conditional expressions, similar to popular programming languages like C# and JavaScript.

The Ternary (?) operator can be very handy when constructing new JSON properties on the fly. For example, now you can write queries to classify the class levels into a human readable form like Beginner/Intermediate/Advanced as shown below.

SELECT (c.grade < 5)? "elementary": "other" AS gradeLevel   
 FROM Families.children[0] c

You can also nest the calls to the operator like in the query below.

SELECT (c.grade < 5)? "elementary": ((c.grade < 9)? "junior": "high") AS gradeLevel   
FROM Families.children[0] c

As with other query operators, if the referenced properties in the conditional expression are missing in any document, or if the types being compared are different, then those documents will be excluded in the query results.

The Coalesce (??) operator can be used to efficiently check for the presence of a property (a.k.a. is defined) in a document. This is useful when querying against semi-structured or data of mixed types. For example, this query returns the “lastName” if present, or the “surname” if it isn’t present.

SELECT f.lastName ?? f.surname AS familyName  
FROM Families f

### Quoted property accessor

You can also access properties using the quoted property operator []. For example, SELECT c.grade and SELECT c["grade"] are equivalent. This syntax is useful when you need to escape a property that contains spaces, special characters, or happens to share the same name as a SQL keyword or reserved word.

SELECT f["lastName"]  
FROM Families f  
WHERE f["id"] = "AndersenFamily"

## SELECT clause

The SELECT clause (**SELECT <select\_list>**) is mandatory and specifies what values will be retrieved from the query, just like in ANSI-SQL. The subset that’s been filtered on top of the source documents are passed onto the projection phase, where the specified JSON values are retrieved and a new JSON object is constructed, for each input passed onto it.

The following example shows a typical SELECT query.

**Query**

SELECT f.address  
FROM Families f   
WHERE f.id = "AndersenFamily"

**Results**

[{  
 "address": {  
 "state": "WA",   
 "county": "King",   
 "city": "seattle"  
 }  
}]

### Nested properties

In the following example, we are projecting two nested properties f.address.state and f.address.city.

**Query**

SELECT f.address.state, f.address.city  
FROM Families f   
WHERE f.id = "AndersenFamily"

**Results**

[{  
 "state": "WA",   
 "city": "seattle"  
}]

Projection also supports JSON expressions as shown in the following example.

**Query**

SELECT { "state": f.address.state, "city": f.address.city, "name": f.id }  
FROM Families f   
WHERE f.id = "AndersenFamily"

**Results**

[{  
 "$1": {  
 "state": "WA",   
 "city": "seattle",   
 "name": "AndersenFamily"  
 }  
}]

Let’s look at the role of $1 here. The SELECT clause needs to create a JSON object and since no key is provided, we use implicit argument variable names starting with $1. For example, this query returns two implicit argument variables, labeled $1 and $2.

**Query**

SELECT { "state": f.address.state, "city": f.address.city },   
 { "name": f.id }  
FROM Families f   
WHERE f.id = "AndersenFamily"

**Results**

[{  
 "$1": {  
 "state": "WA",   
 "city": "seattle"  
 },   
 "$2": {  
 "name": "AndersenFamily"  
 }  
}]

### Aliasing

Now let’s extend the example above with explicit aliasing of values. AS is the keyword used for aliasing. Note that it’s optional as shown while projecting the second value as NameInfo.

In case a query has two properties with the same name, aliasing must be used to rename one or both of the properties so that they are disambiguated in the projected result.

**Query**

SELECT   
 { "state": f.address.state, "city": f.address.city } AS AddressInfo,   
 { "name": f.id } NameInfo  
FROM Families f   
WHERE f.id = "AndersenFamily"

**Results**

[{  
 "AddressInfo": {  
 "state": "WA",   
 "city": "seattle"  
 },   
 "NameInfo": {  
 "name": "AndersenFamily"  
 }  
}]

### Scalar expressions

In addition to property references, the SELECT clause also supports scalar expressions like constants, arithmetic expressions, logical expressions, etc. For example, here’s a simple “Hello World” query.

**Query**

SELECT "Hello World"

**Results**

[{  
 "$1": "Hello World"  
}]

Here’s a more complex example that uses a scalar expression.

**Query**

SELECT ((2 + 11 % 7)-2)/3

**Results**

[{  
 "$1": 1.33333  
}]

In the following example, the result of the scalar expression is a Boolean.

**Query**

SELECT f.address.city = f.address.state AS AreFromSameCityState  
FROM Families f

**Results**

[  
 {  
 "AreFromSameCityState": false  
 },   
 {  
 "AreFromSameCityState": true  
 }  
]

### Object and array creation

Another key feature of DocumentDB SQL is array/object creation. In the previous example, note that we created a new JSON object. Similarly, one can also construct arrays as shown in the following examples.

**Query**

SELECT [f.address.city, f.address.state] AS CityState   
FROM Families f

**Results**

[  
 {  
 "CityState": [  
 "seattle",   
 "WA"  
 ]  
 },   
 {  
 "CityState": [  
 "NY",   
 "NY"  
 ]  
 }  
]

### VALUE keyword

The **VALUE** keyword provides a way to return JSON value. For example, the query shown below returns the scalar "Hello World" instead of {$1: "Hello World"}.

**Query**

SELECT VALUE "Hello World"

**Results**

[  
 "Hello World"  
]

The following query returns the JSON value without the "address" label in the results.

**Query**

SELECT VALUE f.address  
FROM Families f

**Results**

[  
 {  
 "state": "WA",   
 "county": "King",   
 "city": "seattle"  
 },   
 {  
 "state": "NY",   
 "county": "Manhattan",   
 "city": "NY"  
 }  
]

The following example extends this to show how to return JSON primitive values (the leaf level of the JSON tree).

**Query**

SELECT VALUE f.address.state  
FROM Families f

**Results**

[  
 "WA",  
 "NY"  
]

### \* Operator

The special operator (\*) is supported to project the document as-is. When used, it must be the only projected field. While a query like SELECT \* FROM Families f is valid, SELECT VALUE \* FROM Families f and SELECT \*, f.id FROM Families f are not valid.

**Query**

SELECT \*   
FROM Families f   
WHERE f.id = "AndersenFamily"

**Results**

[{  
 "id": "AndersenFamily",  
 "lastName": "Andersen",  
 "parents": [  
 { "firstName": "Thomas" },  
 { "firstName": "Mary Kay"}  
 ],  
 "children": [  
 {  
 "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 }  
 ],  
 "address": { "state": "WA", "county": "King", "city": "seattle" },  
 "creationDate": 1431620472,  
 "isRegistered": true  
}]

### TOP Operator

The TOP keyword can be used to limit the number of values from a query. When TOP is used in conjunction with the ORDER BY clause, the result set is limited to the first N number of ordered values; otherwise, it returns the first N number of results in an undefined order. As a best practice, in a SELECT statement, always use an ORDER BY clause with the TOP clause. This is the only way to predictably indicate which rows are affected by TOP.

**Query**

SELECT TOP 1 \*   
FROM Families f

**Results**

[{  
 "id": "AndersenFamily",  
 "lastName": "Andersen",  
 "parents": [  
 { "firstName": "Thomas" },  
 { "firstName": "Mary Kay"}  
 ],  
 "children": [  
 {  
 "firstName": "Henriette Thaulow", "gender": "female", "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 }  
 ],  
 "address": { "state": "WA", "county": "King", "city": "seattle" },  
 "creationDate": 1431620472,  
 "isRegistered": true  
}]

TOP can be used with a constant value (as shown above) or with a variable value using parameterized queries. For more details, please see parameterized queries below.

## ORDER BY clause

Like in ANSI-SQL, you can include an optional Order By clause while querying. The clause can include an optional ASC/DESC argument to specify the order in which results must be retrieved. For a more detailed look at Order By, refer to [DocumentDB Order By Walkthrough](/documentation/articles/documentdb-orderby).

For example, here’s a query that retrieves families in order of the resident city’s name.

**Query**

SELECT f.id, f.address.city  
FROM Families f   
ORDER BY f.address.city

**Results**

[  
 {  
 "id": "WakefieldFamily",  
 "city": "NY"  
 },  
 {  
 "id": "AndersenFamily",  
 "city": "Seattle"   
 }  
]

And here’s a query that retrieves families in order of creation date, which is stored as a number representing the epoch time, i.e, elapsed time since Jan 1, 1970 in seconds.

**Query**

SELECT f.id, f.creationDate  
FROM Families f   
ORDER BY f.creationDate DESC

**Results**

[  
 {  
 "id": "WakefieldFamily",  
 "creationDate": 1431620462  
 },  
 {  
 "id": "AndersenFamily",  
 "creationDate": 1431620472   
 }  
]

## Advanced database concepts and SQL queries

### Iteration

A new construct was added via the **IN** keyword in DocumentDB SQL to provide support for iterating over JSON arrays. The FROM source provides support for iteration. Let’s start with the following example:

**Query**

SELECT \*   
FROM Families.children

**Results**

[  
 [  
 {  
 "firstName": "Henriette Thaulow",   
 "gender": "female",   
 "grade": 5,   
 "pets": [{ "givenName": "Fluffy"}]  
 }  
 ],   
 [  
 {  
 "familyName": "Merriam",   
 "givenName": "Jesse",   
 "gender": "female",   
 "grade": 1  
 },   
 {  
 "familyName": "Miller",   
 "givenName": "Lisa",   
 "gender": "female",   
 "grade": 8  
 }  
 ]  
]

Now let’s look at another query that performs iteration over children in the collection. Note the difference in the output array. This example splits children and flattens the results into a single array.

**Query**

SELECT \*   
FROM c IN Families.children

**Results**

[  
 {  
 "firstName": "Henriette Thaulow",  
 "gender": "female",  
 "grade": 5,  
 "pets": [{ "givenName": "Fluffy" }]  
 },  
 {  
 "familyName": "Merriam",  
 "givenName": "Jesse",  
 "gender": "female",  
 "grade": 1  
 },  
 {  
 "familyName": "Miller",  
 "givenName": "Lisa",  
 "gender": "female",  
 "grade": 8  
 }  
]

This can be further used to filter on each individual entry of the array as shown in the following example.

**Query**

SELECT c.givenName  
FROM c IN Families.children  
WHERE c.grade = 8

**Results**

[{  
 "givenName": "Lisa"  
}]

### Joins

In a relational database, the need to join across tables is very important. It’s the logical corollary to designing normalized schemas. Contrary to this, DocumentDB deals with the denormalized data model of schema-free documents. This is the logical equivalent of a “self-join”.

The syntax that the language supports is JOIN JOIN … JOIN . Overall, this returns a set of **N**-tuples (tuple with **N** values). Each tuple has values produced by iterating all collection aliases over their respective sets. In other words, this is a full cross product of the sets participating in the join.

The following examples show how the JOIN clause works. In the following example, the result is empty since the cross product of each document from source and an empty set is empty.

**Query**

SELECT f.id  
FROM Families f  
JOIN f.NonExistent

**Results**

[{  
}]

In the following example, the join is between the document root and the children sub-root. It’s a cross product between two JSON objects. The fact that children is an array is not effective in the JOIN since we are dealing with a single root that is the children array. Hence the result contains only two results, since the cross product of each document with the array yields exactly only one document.

**Query**

SELECT f.id  
FROM Families f  
JOIN f.children

**Results**

[  
 {  
 "id": "AndersenFamily"  
 },   
 {  
 "id": "WakefieldFamily"  
 }  
]

The following example shows a more conventional join:

**Query**

SELECT f.id  
FROM Families f  
JOIN c IN f.children

**Results**

[  
 {  
 "id": "AndersenFamily"  
 },   
 {  
 "id": "WakefieldFamily"  
 },   
 {  
 "id": "WakefieldFamily"  
 }  
]

The first thing to note is that the from\_source of the **JOIN** clause is an iterator. So, the flow in this case is as follows:

* Expand each child element **c** in the array.
* Apply a cross product with the root of the document **f** with each child element **c** that was flattened in the first step.
* Finally, project the root object **f** name property alone.

The first document (AndersenFamily) contains only one child element, so the result set contains only a single object corresponding to this document. The second document (WakefieldFamily) contains two children. So, the cross product produces a separate object for each child, thereby resulting in two objects, one for each child corresponding to this document. Note that the root fields in both these documents will be same, just as you would expect in a cross product.

The real utility of the JOIN is to form tuples from the cross-product in a shape that’s otherwise difficult to project. Furthermore, as we will see in the example below, you could filter on the combination of a tuple that lets’ the user chose a condition satisfied by the tuples overall.

**Query**

SELECT   
 f.id AS familyName,  
 c.givenName AS childGivenName,  
 c.firstName AS childFirstName,  
 p.givenName AS petName   
FROM Families f   
JOIN c IN f.children   
JOIN p IN c.pets

**Results**

[  
 {  
 "familyName": "AndersenFamily",   
 "childFirstName": "Henriette Thaulow",   
 "petName": "Fluffy"  
 },   
 {  
 "familyName": "WakefieldFamily",   
 "childGivenName": "Jesse",   
 "petName": "Goofy"  
 },   
 {  
 "familyName": "WakefieldFamily",   
 "childGivenName": "Jesse",   
 "petName": "Shadow"  
 }  
]

This example is a natural extension of the preceding example, and performs a double join. So, the cross product can be viewed as the following pseudo-code.

for-each(Family f in Families)  
{   
 for-each(Child c in f.children)  
 {  
 for-each(Pet p in c.pets)  
 {  
 return (Tuple(f.id AS familyName,   
 c.givenName AS childGivenName,   
 c.firstName AS childFirstName,  
 p.givenName AS petName));  
 }  
 }  
}

AndersenFamily has one child who has one pet. So, the cross product yields one row (1*1*1) from this family. WakefieldFamily however has two children, but only one child “Jesse” has pets. Jesse has 2 pets though. Hence the cross product yields 1*1*2 = 2 rows from this family.

In the next example, there is an additional filter on pet. This excludes all the tuples where the pet name is not “Shadow”. Notice that we are able to build tuples from arrays, filter on any of the elements of the tuple, and project any combination of the elements.

**Query**

SELECT   
 f.id AS familyName,  
 c.givenName AS childGivenName,  
 c.firstName AS childFirstName,  
 p.givenName AS petName   
FROM Families f   
JOIN c IN f.children   
JOIN p IN c.pets  
WHERE p.givenName = "Shadow"

**Results**

[  
 {  
 "familyName": "WakefieldFamily",   
 "childGivenName": "Jesse",   
 "petName": "Shadow"  
 }  
]

## JavaScript integration

DocumentDB provides a programming model for executing JavaScript based application logic directly on the collections in terms of stored procedures and triggers. This allows for both:

* Ability to do high performance transactional CRUD operations and queries against documents in a collection by virtue of the deep integration of JavaScript runtime directly within the database engine.
* A natural modeling of control flow, variable scoping, and assignment and integration of exception handling primitives with database transactions. For more details about DocumentDB support for JavaScript integration, please refer to the JavaScript server side programmability documentation.

### User Defined Functions (UDFs)

Along with the types already defined in this article, DocumentDB SQL provides support for User Defined Functions (UDF). In particular, scalar UDFs are supported where the developers can pass in zero or many arguments and return a single argument result back. Each of these arguments are checked for being legal JSON values.

The DocumentDB SQL syntax is extended to support custom application logic using these User Defined Functions. UDFs can be registered with DocumentDB and then be referenced as part of a SQL query. In fact, the UDFs are exquisitely designed to be invoked by queries. As a corollary to this choice, UDFs do not have access to the context object which the other JavaScript types (stored procedures and triggers) have. Since queries execute as read-only, they can run either on primary or on secondary replicas. Therefore, UDFs are designed to run on secondary replicas unlike other JavaScript types.

Below is an example of how a UDF can be registered at the DocumentDB database, specifically under a document collection.

UserDefinedFunction regexMatchUdf = new UserDefinedFunction  
 {  
 Id = "REGEX\_MATCH",  
 Body = @"function (input, pattern) {   
 return input.match(pattern) !== null;  
 };",  
 };  
   
 UserDefinedFunction createdUdf = client.CreateUserDefinedFunctionAsync(  
 UriFactory.CreateDocumentCollectionUri("testdb", "families"),   
 regexMatchUdf).Result;

The preceding example creates a UDF whose name is REGEX\_MATCH. It accepts two JSON string values input and pattern and checks if the first matches the pattern specified in the second using JavaScript’s string.match() function.

We can now use this UDF in a query in a projection. UDFs must be qualified with the case-sensitive prefix “udf.” when called from within queries.

[AZURE.NOTE] Prior to 3/17/2015, DocumentDB supported UDF calls without the “udf.” prefix like SELECT REGEX\_MATCH(). This calling pattern has been deprecated.

**Query**

SELECT udf.REGEX\_MATCH(Families.address.city, ".\*eattle")  
FROM Families

**Results**

[  
 {  
 "$1": true  
 },   
 {  
 "$1": false  
 }  
]

The UDF can also be used inside a filter as shown in the example below, also qualified with the “udf.” prefix :

**Query**

SELECT Families.id, Families.address.city  
FROM Families  
WHERE udf.REGEX\_MATCH(Families.address.city, ".\*eattle")

**Results**

[{  
 "id": "AndersenFamily",  
 "city": "Seattle"  
}]

In essence, UDFs are valid scalar expressions and can be used in both projections and filters.

To expand on the power of UDFs, let’s look at another example with conditional logic:

UserDefinedFunction seaLevelUdf = new UserDefinedFunction()  
 {  
 Id = "SEALEVEL",  
 Body = @"function(city) {  
 switch (city) {  
 case 'seattle':  
 return 520;  
 case 'NY':  
 return 410;  
 case 'Chicago':  
 return 673;  
 default:  
 return -1;  
 }"  
 };  
  
 UserDefinedFunction createdUdf = await client.CreateUserDefinedFunctionAsync(  
 UriFactory.CreateDocumentCollectionUri("testdb", "families"),   
 seaLevelUdf);

Below is an example that exercises the UDF.

**Query**

SELECT f.address.city, udf.SEALEVEL(f.address.city) AS seaLevel  
FROM Families f

**Results**

[  
 {  
 "city": "seattle",   
 "seaLevel": 520  
 },   
 {  
 "city": "NY",   
 "seaLevel": 410  
 }  
]

As the preceding examples showcase, UDFs integrate the power of JavaScript language with the DocumentDB SQL to provide a rich programmable interface to do complex procedural, conditional logic with the help of inbuilt JavaScript runtime capabilities.

DocumentDB SQL provides the arguments to the UDFs for each document in the source at the current stage (WHERE clause or SELECT clause) of processing the UDF. The result is incorporated in the overall execution pipeline seamlessly. If the properties referred to by the UDF parameters are not available in the JSON value, the parameter is considered as undefined and hence the UDF invocation is entirely skipped. Similarly if the result of the UDF is undefined, it’s not included in the result.

In summary, UDFs are great tools to do complex business logic as part of the query.

### Operator evaluation

DocumentDB, by the virtue of being a JSON database, draws parallels with JavaScript operators and its evaluation semantics. While DocumentDB tries to preserve JavaScript semantics in terms of JSON support, the operation evaluation deviates in some instances.

In DocumentDB SQL, unlike in traditional SQL, the types of values are often not known until the values are actually retrieved from database. In order to efficiently execute queries, most of the operators have strict type requirements.

DocumentDB SQL doesn’t perform implicit conversions, unlike JavaScript. For instance, a query like SELECT \* FROM Person p WHERE p.Age = 21 matches documents which contain an Age property whose value is 21. Any other document whose Age property matches string “21”, or other possibly infinite variations like “021”, “21.0”, “0021”, “00021”, etc. will not be matched. This is in contrast to the JavaScript where the string values are implicitly casted to numbers (based on operator, ex: ==). This choice is crucial for efficient index matching in DocumentDB SQL.

## Parameterized SQL queries

DocumentDB supports queries with parameters expressed with the familiar @ notation. Parameterized SQL provides robust handling and escaping of user input, preventing accidental exposure of data through SQL injection.

For example, you can write a query that takes last name and address state as parameters, and then execute it for various values of last name and address state based on user input.

SELECT \*   
FROM Families f  
WHERE f.lastName = @lastName AND f.address.state = @addressState

This request can then be sent to DocumentDB as a parameterized JSON query like shown below.

{   
 "query": "SELECT \* FROM Families f WHERE f.lastName = @lastName AND f.address.state = @addressState",   
 "parameters": [   
 {"name": "@lastName", "value": "Wakefield"},   
 {"name": "@addressState", "value": "NY"},   
 ]   
}

The argument to TOP can be set using parameterized queries like shown below.

{   
 "query": "SELECT TOP @n \* FROM Families",   
 "parameters": [   
 {"name": "@n", "value": 10},   
 ]   
}

Parameter values can be any valid JSON (strings, numbers, Booleans, null, even arrays or nested JSON). Also since DocumentDB is schema-less, parameters are not validated against any type.

## Built-in functions

DocumentDB also supports a number of built-in functions for common operations, that can be used inside queries like user defined functions (UDFs).

Mathematical functions

ABS, CEILING, EXP, FLOOR, LOG, LOG10, POWER, ROUND, SIGN, SQRT, SQUARE, TRUNC, ACOS, ASIN, ATAN, ATN2, COS, COT, DEGREES, PI, RADIANS, SIN, and TAN

Type checking functions

IS\_ARRAY, IS\_BOOL, IS\_NULL, IS\_NUMBER, IS\_OBJECT, IS\_STRING, IS\_DEFINED, and IS\_PRIMITIVE

String functions

CONCAT, CONTAINS, ENDSWITH, INDEX\_OF, LEFT, LENGTH, LOWER, LTRIM, REPLACE, REPLICATE, REVERSE, RIGHT, RTRIM, STARTSWITH, SUBSTRING, and UPPER

Array functions

ARRAY\_CONCAT, ARRAY\_CONTAINS, ARRAY\_LENGTH, and ARRAY\_SLICE

Spatial functions

ST\_DISTANCE, ST\_WITHIN, ST\_ISVALID, and ST\_ISVALIDDETAILED

If you’re currently using a user defined function (UDF) for which a built-in function is now available, you should use the corresponding built-in function as it is going to be quicker to run and more efficiently.

### Mathematical functions

The mathematical functions each perform a calculation, usually based on input values that are provided as arguments, and return a numeric value. Here’s a table of supported built-in mathematical functions.

Usage

Description

ABS (num\_expr)

Returns the absolute (positive) value of the specified numeric expression.

CEILING (num\_expr)

Returns the smallest integer value greater than, or equal to, the specified numeric expression.

FLOOR (num\_expr)

Returns the largest integer less than or equal to the specified numeric expression.

EXP (num\_expr)

Returns the exponent of the specified numeric expression.

LOG (num\_expr [,base])

Returns the natural logarithm of the specified numeric expression, or the logarithm using the specified base

LOG10 (num\_expr)

Returns the base-10 logarithmic value of the specified numeric expression.

ROUND (num\_expr)

Returns a numeric value, rounded to the closest integer value.

TRUNC (num\_expr)

Returns a numeric value, truncated to the closest integer value.

SQRT (num\_expr)

Returns the square root of the specified numeric expression.

SQUARE (num\_expr)

Returns the square of the specified numeric expression.

POWER (num\_expr, num\_expr)

Returns the power of the specified numeric expression to the value specifed.

SIGN (num\_expr)

Returns the sign value (-1, 0, 1) of the specified numeric expression.

ACOS (num\_expr)

Returns the angle, in radians, whose cosine is the specified numeric expression; also called arccosine.

ASIN (num\_expr)

Returns the angle, in radians, whose sine is the specified numeric expression. This is also called arcsine.

ATAN (num\_expr)

Returns the angle, in radians, whose tangent is the specified numeric expression. This is also called arctangent.

ATN2 (num\_expr)

Returns the angle, in radians, between the positive x-axis and the ray from the origin to the point (y, x), where x and y are the values of the two specified float expressions.

COS (num\_expr)

Returns the trigonometric cosine of the specified angle, in radians, in the specified expression.

COT (num\_expr)

Returns the trigonometric cotangent of the specified angle, in radians, in the specified numeric expression.

DEGREES (num\_expr)

Returns the corresponding angle in degrees for an angle specified in radians.

PI ()

Returns the constant value of PI.

RADIANS (num\_expr)

Returns radians when a numeric expression, in degrees, is entered.

SIN (num\_expr)

Returns the trigonometric sine of the specified angle, in radians, in the specified expression.

TAN (num\_expr)

Returns the tangent of the input expression, in the specified expression.

For example, you can now run queries like the following:

**Query**

SELECT VALUE ABS(-4)

**Results**

[4]

The main difference between DocumentDB’s functions compared to ANSI SQL is that they are designed to work well with schema-less and mixed schema data. For example, if you have a document where the Size property is missing, or has a non-numeric value like “unknown”, then the document is skipped over, instead of returning an error.

### Type checking functions

The type checking functions allow you to check the type of an expression within SQL queries. Type checking functions can be used to determine the type of properties within documents on the fly when it is variable or unknown. Here’s a table of supported built-in type checking functions.

Usage

Description

IS\_ARRAY (expr)

Returns a Boolean indicating if the type of the value is an array.

IS\_BOOL (expr)

Returns a Boolean indicating if the type of the value is a Boolean.

IS\_NULL (expr)

Returns a Boolean indicating if the type of the value is null.

IS\_NUMBER (expr)

Returns a Boolean indicating if the type of the value is a number.

IS\_OBJECT (expr)

Returns a Boolean indicating if the type of the value is a JSON object.

IS\_STRING (expr)

Returns a Boolean indicating if the type of the value is a string.

IS\_DEFINED (expr)

Returns a Boolean indicating if the property has been assigned a value.

IS\_PRIMITIVE (expr)

Returns a Boolean indicating if the type of the value is a string, number, Boolean or null.

Using these functions, you can now run queries like the following:

**Query**

SELECT VALUE IS\_NUMBER(-4)

**Results**

[true]

### String functions

The following scalar functions perform an operation on a string input value and return a string, numeric or Boolean value. Here’s a table of built-in string functions:

|  |  |
| --- | --- |
| Usage | Description |
| [LENGTH (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_length) | Returns the number of characters of the specified string expression |
| [CONCAT (str\_expr, str\_expr [, str\_expr])](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_concat) | Returns a string that is the result of concatenating two or more string values. |
| [SUBSTRING (str\_expr, num\_expr, num\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_substring) | Returns part of a string expression. |
| [STARTSWITH (str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_startswith) | Returns a Boolean indicating whether the first string expression ends with the second |
| [ENDSWITH (str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_endswith) | Returns a Boolean indicating whether the first string expression ends with the second |
| [CONTAINS (str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_contains) | Returns a Boolean indicating whether the first string expression contains the second. |
| [INDEX\_OF (str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_index_of) | Returns the starting position of the first occurrence of the second string expression within the first specified string expression, or -1 if the string is not found. |
| [LEFT (str\_expr, num\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_left) | Returns the left part of a string with the specified number of characters. |
| [RIGHT (str\_expr, num\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_right) | Returns the right part of a string with the specified number of characters. |
| [LTRIM (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_ltrim) | Returns a string expression after it removes leading blanks. |
| [RTRIM (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_rtrim) | Returns a string expression after truncating all trailing blanks. |
| [LOWER (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_lower) | Returns a string expression after converting uppercase character data to lowercase. |
| [UPPER (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_upper) | Returns a string expression after converting lowercase character data to uppercase. |
| [REPLACE (str\_expr, str\_expr, str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_replace) | Replaces all occurrences of a specified string value with another string value. |
| [REPLICATE (str\_expr, num\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_replicate) | Repeats a string value a specified number of times. |
| [REVERSE (str\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_reverse) | Returns the reverse order of a string value. |

Using these functions, you can now run queries like the following. For example, you can return the family name in uppercase as follows:

**Query**

SELECT VALUE UPPER(Families.id)  
FROM Families

**Results**

[  
 "WAKEFIELDFAMILY",   
 "ANDERSENFAMILY"  
]

Or concatenate strings like in this example:

**Query**

SELECT Families.id, CONCAT(Families.address.city, ",", Families.address.state) AS location  
FROM Families

**Results**

[{  
 "id": "WakefieldFamily",  
 "location": "NY,NY"  
},  
{  
 "id": "AndersenFamily",  
 "location": "seattle,WA"  
}]

String functions can also be used in the WHERE clause to filter results, like in the following example:

**Query**

SELECT Families.id, Families.address.city  
FROM Families  
WHERE STARTSWITH(Families.id, "Wakefield")

**Results**

[{  
 "id": "WakefieldFamily",  
 "city": "NY"  
}]

### Array functions

The following scalar functions perform an operation on an array input value and return numeric, Boolean or array value. Here’s a table of built-in array functions:

|  |  |
| --- | --- |
| Usage | Description |
| [ARRAY\_LENGTH (arr\_expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_array_length) | Returns the number of elements of the specified array expression. |
| [ARRAY\_CONCAT (arr\_expr, arr\_expr [, arr\_expr])](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_array_concat) | Returns an array that is the result of concatenating two or more array values. |
| [ARRAY\_CONTAINS (arr\_expr, expr)](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_array_contains) | Returns a Boolean indicating whether the array contains the specified value. |
| [ARRAY\_SLICE (arr\_expr, num\_expr [, num\_expr])](https://msdn.microsoft.com/library/azure/dn782250.aspx#bk_array_slice) | Returns part of an array expression. |

Array functions can be used to manipulate arrays within JSON. For example, here’s a query that returns all documents where one of the parents is “Robin Wakefield”.

**Query**

SELECT Families.id   
FROM Families   
WHERE ARRAY\_CONTAINS(Families.parents, { givenName: "Robin", familyName: "Wakefield" })

**Results**

[{  
 "id": "WakefieldFamily"  
}]

Here’s another example that uses ARRAY\_LENGTH to get the number of children per family.

**Query**

SELECT Families.id, ARRAY\_LENGTH(Families.children) AS numberOfChildren  
FROM Families

**Results**

[{  
 "id": "WakefieldFamily",  
 "numberOfChildren": 2  
},  
{  
 "id": "AndersenFamily",  
 "numberOfChildren": 1  
}]

### Spatial functions

DocumentDB supports the following Open Geospatial Consortium (OGC) built-in functions for geospatial querying. For more details on geospatial support in DocumentDB, please see [Working with geospatial data in Azure DocumentDB](/documentation/articles/documentdb-geospatial).

Usage

Description

ST\_DISTANCE (point\_expr, point\_expr)

Returns the distance between the two GeoJSON point expressions.

ST\_WITHIN (point\_expr, polygon\_expr)

Returns a Boolean expression indicating whether the GeoJSON point specified in the first argument is within the GeoJSON polygon in the second argument.

ST\_ISVALID

Returns a Boolean value indicating whether the specified GeoJSON point or polygon expression is valid.

ST\_ISVALIDDETAILED

Returns a JSON value containing a Boolean value if the specified GeoJSON point or polygon expression is valid, and if invalid, additionally the reason as a string value.

Spatial functions can be used to perform proximity querries against spatial data. For example, here’s a query that returns all family documents that are within 30 km of the specified location using the ST\_DISTANCE built-in function.

**Query**

SELECT f.id   
FROM Families f   
WHERE ST\_DISTANCE(f.location, {'type': 'Point', 'coordinates':[31.9, -4.8]}) < 30000

**Results**

[{  
 "id": "WakefieldFamily"  
}]

If you include spatial indexing in your indexing policy, then “distance queries” will be served efficiently through the index. For more details on spatial indexing, please see the section below. If you don’t have a spatial index for the specified paths, you can still perform spatial queries by specifying x-ms-documentdb-query-enable-scan request header with the value set to “true”. In .NET, this can be done by passing the optional **FeedOptions** argument to queries with [EnableScanInQuery](https://msdn.microsoft.com/library/microsoft.azure.documents.client.feedoptions.enablescaninquery.aspx#P:Microsoft.Azure.Documents.Client.FeedOptions.EnableScanInQuery) set to true.

ST\_WITHIN can be used to check if a point lies within a polygon. Commonly polygons are used to represent boundaries like zip codes, state boundaries, or natural formations. Again if you include spatial indexing in your indexing policy, then “within” queries will be served efficiently through the index.

Polygon arguments in ST\_WITHIN can contain only a single ring, i.e. the polygons must not contain holes in them. Check the [DocumentDB limits](/documentation/articles/documentdb-limits) for the maximum number of points allowed in a polygon for an ST\_WITHIN query.

**Query**

SELECT \*   
FROM Families f   
WHERE ST\_WITHIN(f.location, {  
 'type':'Polygon',   
 'coordinates': [[[31.8, -5], [32, -5], [32, -4.7], [31.8, -4.7], [31.8, -5]]]  
})

**Results**

[{  
 "id": "WakefieldFamily",  
}]

[AZURE.NOTE] Similar to how mismatched types works in DocumentDB query, if the location value specified in either argument is malformed or invalid, then it will evaluate to **undefined** and the evaluated document to be skipped from the query results. If your query returns no results, run ST\_ISVALIDDETAILED To debug why the spatail type is invalid.

ST\_ISVALID and ST\_ISVALIDDETAILED can be used to check if a spatial object is valid. For example, the following query checks the validity of a point with an out of range latitude value (-132.8). ST\_ISVALID returns just a Boolean value, and ST\_ISVALIDDETAILED returns the Boolean and a string containing the reason why it is considered invalid.

**Query**

SELECT ST\_ISVALID({ "type": "Point", "coordinates": [31.9, -132.8] })

**Results**

[{  
 "$1": false  
}]

These functions can also be used to validate polygons. For example, here we use ST\_ISVALIDDETAILED to validate a polygon that is not closed.

**Query**

SELECT ST\_ISVALIDDETAILED({ "type": "Polygon", "coordinates": [[   
 [ 31.8, -5 ], [ 31.8, -4.7 ], [ 32, -4.7 ], [ 32, -5 ]   
 ]]})

**Results**

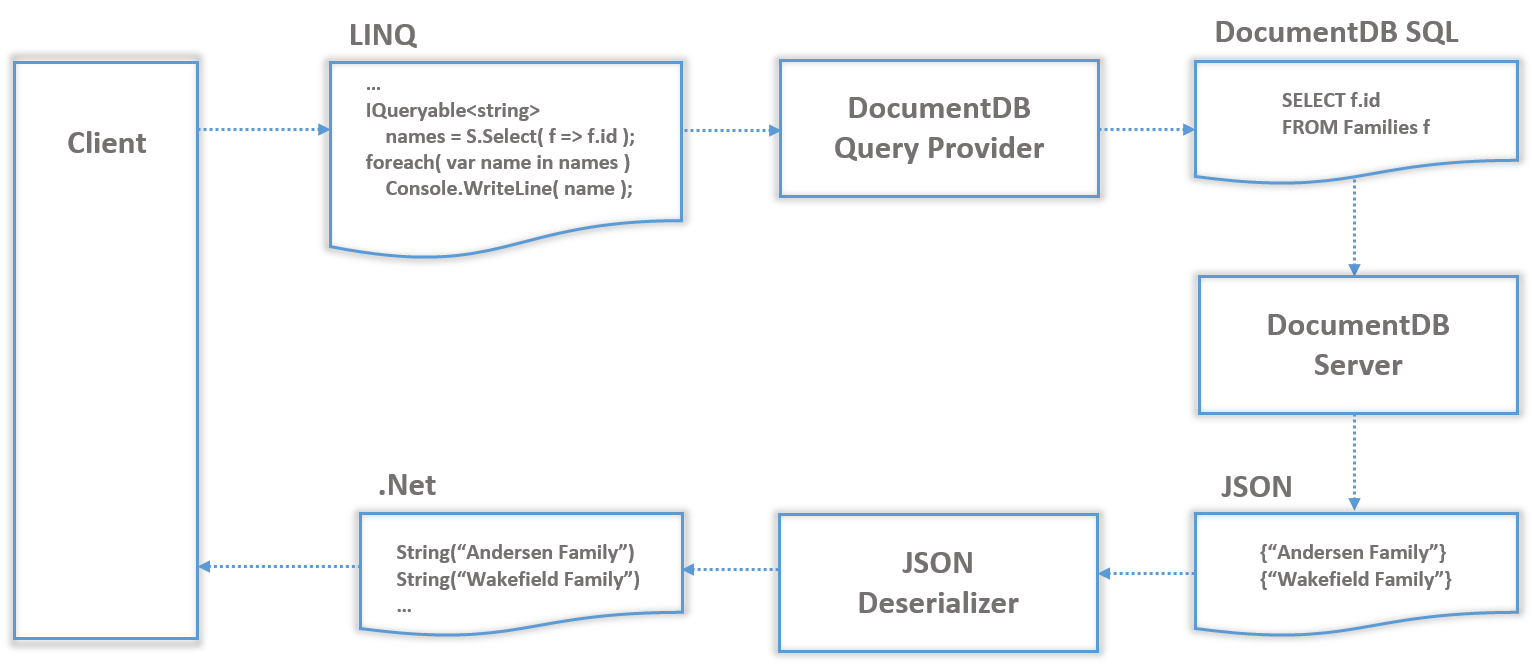
[{  
 "$1": {   
 "valid": false,   
 "reason": "The Polygon input is not valid because the start and end points of the ring number 1 are not the same. Each ring of a polygon must have the same start and end points."   
 }  
}]

That wraps up spatial functions, and the SQL syntax for DocumentDB. Now let’s take a look at how LINQ querying works and how it interacts with the syntax we’ve seen so far.

## LINQ to DocumentDB SQL

LINQ is a .NET programming model that expresses computation as queries on streams of objects. DocumentDB provides a client side library to interface with LINQ by facilitating a conversion between JSON and .NET objects and a mapping from a subset of LINQ queries to DocumentDB queries.

The picture below shows the architecture of supporting LINQ queries using DocumentDB. Using the DocumentDB client, developers can create an **IQueryable** object that directly queries the DocumentDB query provider, which then translates the LINQ query into a DocumentDB query. The query is then passed to the DocumentDB server to retrieve a set of results in JSON format. The returned results are deserialized into a stream of .NET objects on the client side.



Architecture of supporting LINQ queries using DocumentDB - SQL syntax, JSON query language, database concepts and SQL queries

### .NET and JSON mapping

The mapping between .NET objects and JSON documents is natural - each data member field is mapped to a JSON object, where the field name is mapped to the “key” part of the object and the “value” part is recursively mapped to the value part of the object. Consider the following example. The Family object created is mapped to the JSON document as shown below. And vice versa, the JSON document is mapped back to a .NET object.

**C# Class**

public class Family  
{  
 [JsonProperty(PropertyName="id")]  
 public string Id;  
 public Parent[] parents;  
 public Child[] children;  
 public bool isRegistered;  
};  
  
public struct Parent  
{  
 public string familyName;  
 public string givenName;  
};  
  
public class Child  
{  
 public string familyName;  
 public string givenName;  
 public string gender;  
 public int grade;  
 public List<Pet> pets;  
};  
  
public class Pet  
{  
 public string givenName;  
};  
  
public class Address  
{  
 public string state;  
 public string county;  
 public string city;  
};  
  
// Create a Family object.  
Parent mother = new Parent { familyName= "Wakefield", givenName="Robin" };  
Parent father = new Parent { familyName = "Miller", givenName = "Ben" };  
Child child = new Child { familyName="Merriam", givenName="Jesse", gender="female", grade=1 };  
Pet pet = new Pet { givenName = "Fluffy" };  
Address address = new Address { state = "NY", county = "Manhattan", city = "NY" };  
Family family = new Family { Id = "WakefieldFamily", parents = new Parent [] { mother, father}, children = new Child[] { child }, isRegistered = false };

**JSON**

{  
 "id": "WakefieldFamily",  
 "parents": [  
 { "familyName": "Wakefield", "givenName": "Robin" },  
 { "familyName": "Miller", "givenName": "Ben" }  
 ],  
 "children": [  
 {  
 "familyName": "Merriam",   
 "givenName": "Jesse",   
 "gender": "female",   
 "grade": 1,  
 "pets": [  
 { "givenName": "Goofy" },  
 { "givenName": "Shadow" }  
 ]  
 },  
 {   
 "familyName": "Miller",   
 "givenName": "Lisa",   
 "gender": "female",   
 "grade": 8   
 }  
 ],  
 "address": { "state": "NY", "county": "Manhattan", "city": "NY" },  
 "isRegistered": false  
};

### LINQ to SQL translation

The DocumentDB query provider performs a best effort mapping from a LINQ query into a DocumentDB SQL query. In the following description, we assume the reader has a basic familiarity of LINQ.

First, for the type system, we support all JSON primitive types – numeric types, boolean, string, and null. Only these JSON types are supported. The following scalar expressions are supported.

* Constant values – these includes constant values of the primitive data types at the time the query is evaluated.
* Property/array index expressions – these expressions refer to the property of an object or an array element.
* family.Id;  
  family.children[0].familyName;  
  family.children[0].grade;  
  family.children[n].grade; //n is an int variable
* Arithmetic expressions - These include common arithmetic expressions on numerical and boolean values. For the complete list, refer to the SQL specification.
* 2 \* family.children[0].grade;  
  x + y;
* String comparison expression - these include comparing a string value to some constant string value.
* mother.familyName == "Smith";  
  child.givenName == s; //s is a string variable
* Object/array creation expression - these expressions return an object of compound value type or anonymous type or an array of such objects. These values can be nested.
* new Parent { familyName = "Smith", givenName = "Joe" };  
  new { first = 1, second = 2 }; //an anonymous type with 2 fields   
  new int[] { 3, child.grade, 5 };

### List of supported LINQ operators

Here is a list of supported LINQ operators in the LINQ provider included with the DocumentDB .NET SDK.

* **Select**: Projections translate to the SQL SELECT including object construction
* **Where**: Filters translate to the SQL WHERE, and support translation between && , || and ! to the SQL operators
* **SelectMany**: Allows unwinding of arrays to the SQL JOIN clause. Can be used to chain/nest expressions to filter on array elements
* **OrderBy and OrderByDescending**: Translates to ORDER BY ascending/descending:
* **CompareTo**: Translates to range comparisons. Commonly used for strings since they’re not comparable in .NET
* **Take**: Translates to the SQL TOP for limiting results from a query
* **Math Functions**: Supports translation from .NET’s Abs, Acos, Asin, Atan, Ceiling, Cos, Exp, Floor, Log, Log10, Pow, Round, Sign, Sin, Sqrt, Tan, Truncate to the equivalent SQL built-in functions.
* **String Functions**: Supports translation from .NET’s Concat, Contains, EndsWith, IndexOf, Count, ToLower, TrimStart, Replace, Reverse, TrimEnd, StartsWith, SubString, ToUpper to the equivalent SQL built-in functions.
* **Array Functions**: Supports translation from .NET’s Concat, Contains, and Count to the equivalent SQL built-in functions.
* **Geospatial Extension Functions**: Supports translation from stub methods Distance, Within, IsValid, and IsValidDetailed to the equivalent SQL built-in functions.
* **User Defined Function Extension Function**: Supports translation from the stub method UserDefinedFunctionProvider.Invoke to the corresponding user defined function.
* **Miscellaneous**: Supports translation of the coalesce and conditional operators. Can translate Contains to String CONTAINS, ARRAY\_CONTAINS or the SQL IN depending on context.

### SQL query operators

Here are some examples that illustrate how some of the standard LINQ query operators are translated down to DocumentDB queries.

#### Select Operator

The syntax is input.Select(x => f(x)), where f is a scalar expression.

**LINQ lambda expression**

input.Select(family => family.parents[0].familyName);

**SQL**

SELECT VALUE f.parents[0].familyName  
FROM Families f

**LINQ lambda expression**

input.Select(family => family.children[0].grade + c); // c is an int variable

**SQL**

SELECT VALUE f.children[0].grade + c  
FROM Families f

**LINQ lambda expression**

input.Select(family => new  
{  
 name = family.children[0].familyName,  
 grade = family.children[0].grade + 3  
});

**SQL**

SELECT VALUE {"name":f.children[0].familyName,   
 "grade": f.children[0].grade + 3 }  
FROM Families f

#### SelectMany operator

The syntax is input.SelectMany(x => f(x)), where f is a scalar expression that returns a collection type.

**LINQ lambda expression**

input.SelectMany(family => family.children);

**SQL**

SELECT VALUE child  
FROM child IN Families.children

#### Where operator

The syntax is input.Where(x => f(x)), where f is a scalar expression which returns a Boolean value.

**LINQ lambda expression**

input.Where(family=> family.parents[0].familyName == "Smith");

**SQL**

SELECT \*  
FROM Families f  
WHERE f.parents[0].familyName = "Smith"

**LINQ lambda expression**

input.Where(  
 family => family.parents[0].familyName == "Smith" &&   
 family.children[0].grade < 3);

**SQL**

SELECT \*  
FROM Families f  
WHERE f.parents[0].familyName = "Smith"  
AND f.children[0].grade < 3

### Composite SQL queries

The above operators can be composed to form more powerful queries. Since DocumentDB supports nested collections, the composition can either be concatenated or nested.

#### Concatenation

The syntax is input(.|.SelectMany())(.Select()|.Where())\*. A concatenated query can start with an optional SelectMany query followed by multiple Select or Where operators.

**LINQ lambda expression**

input.Select(family=>family.parents[0])  
 .Where(familyName == "Smith");

**SQL**

SELECT \*  
FROM Families f  
WHERE f.parents[0].familyName = "Smith"

**LINQ lambda expression**

input.Where(family => family.children[0].grade > 3)  
 .Select(family => family.parents[0].familyName);

**SQL**

SELECT VALUE f.parents[0].familyName  
FROM Families f  
WHERE f.children[0].grade > 3

**LINQ lambda expression**

input.Select(family => new { grade=family.children[0].grade}).  
 Where(anon=> anon.grade < 3);

**SQL**

SELECT \*  
FROM Families f  
WHERE ({grade: f.children[0].grade}.grade > 3)

**LINQ lambda expression**

input.SelectMany(family => family.parents)  
 .Where(parent => parents.familyName == "Smith");

**SQL**

SELECT \*  
FROM p IN Families.parents  
WHERE p.familyName = "Smith"

#### Nesting

The syntax is input.SelectMany(x=>x.Q()) where Q is a Select, SelectMany, or Where operator.

In a nested query, the inner query is applied to each element of the outer collection. One important feature is that the inner query can refer to the fields of the elements in the outer collection like self-joins.

**LINQ lambda expression**

input.SelectMany(family=>   
 family.parents.Select(p => p.familyName));

**SQL**

SELECT VALUE p.familyName  
FROM Families f  
JOIN p IN f.parents

**LINQ lambda expression**

input.SelectMany(family =>   
 family.children.Where(child => child.familyName == "Jeff"));

**SQL**

SELECT \*  
FROM Families f  
JOIN c IN f.children  
WHERE c.familyName = "Jeff"

**LINQ lambda expression**

input.SelectMany(family => family.children.Where(  
 child => child.familyName == family.parents[0].familyName));

**SQL**

SELECT \*  
FROM Families f  
JOIN c IN f.children  
WHERE c.familyName = f.parents[0].familyName

## Executing SQL queries

DocumentDB exposes resources through a REST API that can be called by any language capable of making HTTP/HTTPS requests. Additionally, DocumentDB offers programming libraries for several popular languages like .NET, Node.js, JavaScript and Python. The REST API and the various libraries all support querying through SQL. The .NET SDK supports LINQ querying in addition to SQL.

The following examples show how to create a query and submit it against a DocumentDB database account.

### REST API

DocumentDB offers an open RESTful programming model over HTTP. Database accounts can be provisioned using an Azure subscription. The DocumentDB resource model consists of a sets of resources under a database account, each of which is addressable using a logical and stable URI. A set of resources is referred to as a feed in this document. A database account consists of a set of databases, each containing multiple collections, each of which in-turn contain documents, UDFs, and other resource types.

The basic interaction model with these resources is through the HTTP verbs GET, PUT, POST and DELETE with their standard interpretation. The POST verb is used for creation of a new resource, for executing a stored procedure or for issuing a DocumentDB query. Queries are always read only operations with no side-effects.

The following examples show a POST for a DocumentDB query made against a collection containing the two sample documents we’ve reviewed so far. The query has a simple filter on the JSON name property. Note the use of the x-ms-documentdb-isquery and Content-Type: application/query+json headers to denote that the operation is a query.

**Request**

POST https://<REST URI>/docs HTTP/1.1  
...  
x-ms-documentdb-isquery: True  
Content-Type: application/query+json  
  
{   
 "query": "SELECT \* FROM Families f WHERE f.id = @familyId",   
 "parameters": [   
 {"name": "@familyId", "value": "AndersenFamily"}   
 ]   
}

**Results**

HTTP/1.1 200 Ok  
x-ms-activity-id: 8b4678fa-a947-47d3-8dd3-549a40da6eed  
x-ms-item-count: 1  
x-ms-request-charge: 0.32  
  
<indented for readability, results highlighted>  
  
{   
 "\_rid":"u1NXANcKogE=",  
 "Documents":[   
 {   
 "id":"AndersenFamily",  
 "lastName":"Andersen",  
 "parents":[   
 {   
 "firstName":"Thomas"  
 },  
 {   
 "firstName":"Mary Kay"  
 }  
 ],  
 "children":[   
 {   
 "firstName":"Henriette Thaulow",  
 "gender":"female",  
 "grade":5,  
 "pets":[   
 {   
 "givenName":"Fluffy"  
 }  
 ]  
 }  
 ],  
 "address":{   
 "state":"WA",  
 "county":"King",  
 "city":"seattle"  
 },  
 "\_rid":"u1NXANcKogEcAAAAAAAAAA==",  
 "\_ts":1407691744,  
 "\_self":"dbs\/u1NXAA==\/colls\/u1NXANcKogE=\/docs\/u1NXANcKogEcAAAAAAAAAA==\/",  
 "\_etag":"00002b00-0000-0000-0000-53e7abe00000",  
 "\_attachments":"\_attachments\/"  
 }  
 ],  
 "count":1  
}

The second example shows a more complex query that returns multiple results from the join.

**Request**

POST https://<REST URI>/docs HTTP/1.1  
...  
x-ms-documentdb-isquery: True  
Content-Type: application/query+json  
  
{   
 "query": "SELECT   
 f.id AS familyName,   
 c.givenName AS childGivenName,   
 c.firstName AS childFirstName,   
 p.givenName AS petName   
 FROM Families f   
 JOIN c IN f.children   
 JOIN p in c.pets",   
 "parameters": []   
}

**Results**

HTTP/1.1 200 Ok  
x-ms-activity-id: 568f34e3-5695-44d3-9b7d-62f8b83e509d  
x-ms-item-count: 1  
x-ms-request-charge: 7.84  
  
<indented for readability, results highlighted>  
  
{   
 "\_rid":"u1NXANcKogE=",  
 "Documents":[   
 {   
 "familyName":"AndersenFamily",  
 "childFirstName":"Henriette Thaulow",  
 "petName":"Fluffy"  
 },  
 {   
 "familyName":"WakefieldFamily",  
 "childGivenName":"Jesse",  
 "petName":"Goofy"  
 },  
 {   
 "familyName":"WakefieldFamily",  
 "childGivenName":"Jesse",  
 "petName":"Shadow"  
 }  
 ],  
 "count":3  
}

If a query’s results cannot fit within a single page of results, then the REST API returns a continuation token through the x-ms-continuation-token response header. Clients can paginate results by including the header in subsequent results. The number of results per page can also be controlled through the x-ms-max-item-count number header.

To manage the data consistency policy for queries, use the x-ms-consistency-level header like all REST API requests. For session consistency, it is required to also echo the latest x-ms-session-token Cookie header in the query request. Note that the queried collection’s indexing policy can also influence the consistency of query results. With the default indexing policy settings, for collections the index is always current with the document contents and query results will match the consistency chosen for data. If the indexing policy is relaxed to Lazy, then queries can return stale results. For more information, refer to [DocumentDB Consistency Levels](/documentation/articles/documentdb-consistency-levels).

If the configured indexing policy on the collection cannot support the specified query, the DocumentDB server returns 400 “Bad Request”. This is returned for range queries against paths configured for hash (equality) lookups, and for paths explicitly excluded from indexing. The x-ms-documentdb-query-enable-scan header can be specified to allow the query to perform a scan when an index is not available.

### C# (.NET) SDK

The .NET SDK supports both LINQ and SQL querying. The following example shows how to perform the simple filter query introduced earlier in this document.

foreach (var family in client.CreateDocumentQuery(collectionLink,   
 "SELECT \* FROM Families f WHERE f.id = \"AndersenFamily\""))  
{  
 Console.WriteLine("\tRead {0} from SQL", family);  
}  
  
SqlQuerySpec query = new SqlQuerySpec("SELECT \* FROM Families f WHERE f.id = @familyId");  
query.Parameters = new SqlParameterCollection();  
query.Parameters.Add(new SqlParameter("@familyId", "AndersenFamily"));  
  
foreach (var family in client.CreateDocumentQuery(collectionLink, query))  
{  
 Console.WriteLine("\tRead {0} from parameterized SQL", family);  
}  
  
foreach (var family in (  
 from f in client.CreateDocumentQuery(collectionLink)  
 where f.Id == "AndersenFamily"  
 select f))  
{  
 Console.WriteLine("\tRead {0} from LINQ query", family);  
}  
  
foreach (var family in client.CreateDocumentQuery(collectionLink)  
 .Where(f => f.Id == "AndersenFamily")  
 .Select(f => f))  
{  
 Console.WriteLine("\tRead {0} from LINQ lambda", family);  
}

This sample compares two properties for equality within each document and uses anonymous projections.

foreach (var family in client.CreateDocumentQuery(collectionLink,  
 @"SELECT {""Name"": f.id, ""City"":f.address.city} AS Family   
 FROM Families f   
 WHERE f.address.city = f.address.state"))  
{  
 Console.WriteLine("\tRead {0} from SQL", family);  
}  
  
foreach (var family in (  
 from f in client.CreateDocumentQuery<Family>(collectionLink)  
 where f.address.city == f.address.state  
 select new { Name = f.Id, City = f.address.city }))  
{  
 Console.WriteLine("\tRead {0} from LINQ query", family);  
}  
  
foreach (var family in  
 client.CreateDocumentQuery<Family>(collectionLink)  
 .Where(f => f.address.city == f.address.state)  
 .Select(f => new { Name = f.Id, City = f.address.city }))  
{  
 Console.WriteLine("\tRead {0} from LINQ lambda", family);  
}

The next sample shows joins, expressed through LINQ SelectMany.

foreach (var pet in client.CreateDocumentQuery(collectionLink,  
 @"SELECT p  
 FROM Families f   
 JOIN c IN f.children   
 JOIN p in c.pets   
 WHERE p.givenName = ""Shadow"""))  
{  
 Console.WriteLine("\tRead {0} from SQL", pet);  
}  
  
// Equivalent in Lambda expressions  
foreach (var pet in  
 client.CreateDocumentQuery<Family>(collectionLink)  
 .SelectMany(f => f.children)  
 .SelectMany(c => c.pets)  
 .Where(p => p.givenName == "Shadow"))  
{  
 Console.WriteLine("\tRead {0} from LINQ lambda", pet);  
}

The .NET client automatically iterates through all the pages of query results in the foreach blocks as shown above. The query options introduced in the REST API section are also available in the .NET SDK using the FeedOptions and FeedResponse classes in the CreateDocumentQuery method. The number of pages can be controlled using the MaxItemCount setting.

You can also explicitly control paging by creating IDocumentQueryable using the IQueryable object, then by reading theResponseContinuationToken values and passing them back as RequestContinuationToken in FeedOptions. EnableScanInQuery can be set to enable scans when the query cannot be supported by the configured indexing policy. For partitioned collections, you can use PartitionKey to run the query against a single partition (though DocumentDB can automatically extract this from the query text), and EnableCrossPartitionQuery to run queries that may need to be run against multiple partitions.

Refer to [DocumentDB .NET samples](https://github.com/Azure/azure-documentdb-net) for more samples containing queries.

### JavaScript server-side API

DocumentDB provides a programming model for executing JavaScript based application logic directly on the collections using stored procedures and triggers. The JavaScript logic registered at a collection level can then issue database operations on the operations on the documents of the given collection. These operations are wrapped in ambient ACID transactions.

The following example show how to use the queryDocuments in the JavaScript server API to make queries from inside stored procedures and triggers.

function businessLogic(name, author) {  
 var context = getContext();  
 var collectionManager = context.getCollection();  
 var collectionLink = collectionManager.getSelfLink()  
  
 // create a new document.  
 collectionManager.createDocument(collectionLink,  
 { name: name, author: author },  
 function (err, documentCreated) {  
 if (err) throw new Error(err.message);  
  
 // filter documents by author  
 var filterQuery = "SELECT \* from root r WHERE r.author = 'George R.'";  
 collectionManager.queryDocuments(collectionLink,  
 filterQuery,  
 function (err, matchingDocuments) {  
 if (err) throw new Error(err.message);  
context.getResponse().setBody(matchingDocuments.length);  
  
 // Replace the author name for all documents that satisfied the query.  
 for (var i = 0; i < matchingDocuments.length; i++) {  
 matchingDocuments[i].author = "George R. R. Martin";  
 // we don't need to execute a callback because they are in parallel  
 collectionManager.replaceDocument(matchingDocuments[i].\_self,  
 matchingDocuments[i]);  
 }  
 })  
 });  
}

## References

1. [Introduction to Azure DocumentDB](/documentation/articles/documentdb-introduction)
2. [DocumentDB SQL specification](http://go.microsoft.com/fwlink/p/?LinkID=510612)
3. [DocumentDB .NET samples](https://github.com/Azure/azure-documentdb-net)
4. [DocumentDB Consistency Levels](/documentation/articles/documentdb-consistency-levels)
5. ANSI SQL 2011 <http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=53681>
6. JSON <http://json.org/>
7. Javascript Specification <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
8. LINQ <http://msdn.microsoft.com/library/bb308959.aspx>
9. Query evaluation techniques for large databases <http://dl.acm.org/citation.cfm?id=152611>
10. Query Processing in Parallel Relational Database Systems, IEEE Computer Society Press, 1994
11. Lu, Ooi, Tan, Query Processing in Parallel Relational Database Systems, IEEE Computer Society Press, 1994.
12. Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins: Pig Latin: A Not-So-Foreign Language for Data Processing, SIGMOD 2008.
13. G. Graefe. The Cascades framework for query optimization. IEEE Data Eng. Bull., 18(3): 1995.