DocumentDB SQL

tutorialspoint

SIMPLYEASYLEARNING

About the Tutorial

DocumentDB is Microsoft's newest NoSQL document database platform that runs on Azure. DocumentDB is designed keeping in mind the requirements of managing data for latest applications. This tutorial talks about querying documents using the special version of SQL supported by DocumentDB with illustrative examples.

Audience

This tutorial is designed for developers who want to get acquainted with how to query DocumentDB using a familiar Structured Query Language (SQL).

Prerequisites

It is an elementary tutorial that explains the basics of DocumentDB and there are no prerequisites as such. However, it will certainly help if you have some prior exposure to NoSQL technologies.

Disclaimer & Copyright

Table of Contents

1. DocumentDB SQL – Overview

DocumentDB is Microsoft's newest NoSQL document database platform that runs on Azure. In this tutorial, we will learn all about querying documents using the special version of SQL supported by DocumentDB.

NoSQL Document Database

DocumentDB is Microsoft's newest NoSQL document database, however, when we say NoSQL document database, what precisely do we mean by NoSQL, and document database?

- SQL means Structured Query Language which is a traditional query language of relational databases. SQL is often equated with relational databases.

- It is really more helpful to think of a NoSQL database as a non-relational database, so NoSQL really means non-relational.

There are different types of NoSQL databases which include key value stores such as:

- Azure Table Storage
- Column-based stores, like Cassandra
- Graph databases, like NEO4
- Document databases, like MongoDB and Azure DocumentDB

Why SQL Syntax?

This can sound strange at first, but in DocumentDB which is a NoSQL database, we query using SQL. As mentioned above, this is a special version of SQL rooted in JSON and JavaScript semantics.

- SQL is just a language, but it's also a very popular language that's rich and expressive. Thus, it definitely seems like a good idea to use some dialect of SQL rather than come up with a whole new way of expressing queries that we would need to learn if you wanted to get documents out of your database.

- SQL is designed for relational databases, and DocumentDB is a non-relational document database. DocumentDB team has actually adapted the SQL syntax for the non-relational world of document databases, and this is what is meant by rooting SQL in JSON and JavaScript.

- The language still reads as familiar SQL, but the semantics are all based on schema-free JSON documents rather than relational tables. In DocumentDB, we will be working with JavaScript data types rather than SQL data types. We will be familiar with SELECT, FROM, WHERE, and so on, but with JavaScript types, which are limited to numbers and strings, objects, arrays, Boolean, and null are far fewer than the wide range of SQL data types.

- Similarly, expressions are evaluated as JavaScript expressions rather than some form of T-SQL. For example, in a world of denormalized data, we're not dealing with the rows and columns, but schema-free documents with hierarchal structures that contain nested arrays and objects.

How does SQL Work?

The DocumentDB team has answered this question in several innovative ways. Few of them are listed as follows:

- First, assuming you've not changed the default behavior to automatically index every property in a document, you can use dotted notation in your queries to navigate a path to any property no matter how deeply nested it may be within the document.

- You can also perform an intra-document join in which nested array elements are joined with their parent element within a document in a manner very similar to the way a join is performed between two tables in the relational world.

- Your queries can return documents from the database as it is, or you can project any custom JSON shape you want based on as much or as little of the document data that you want.

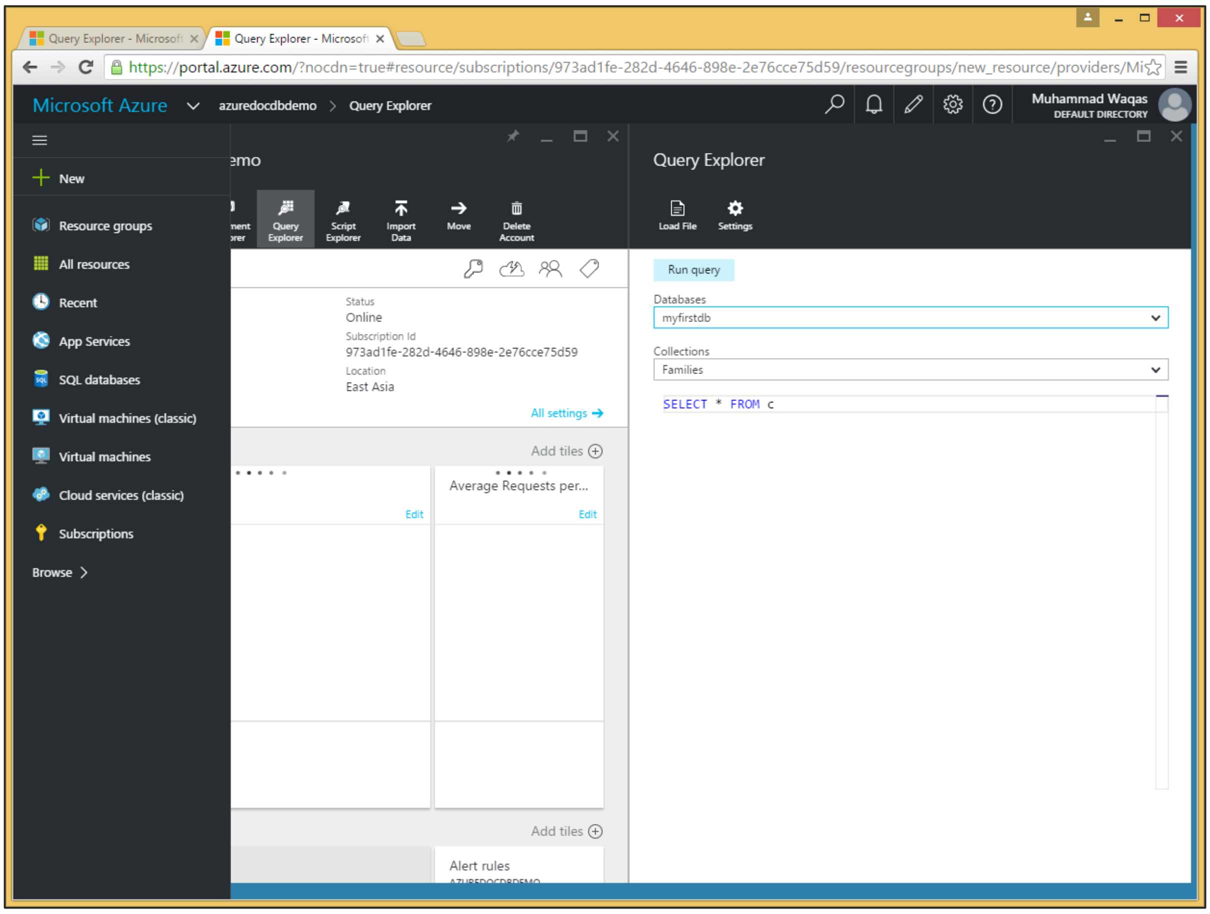
- SQL in DocumentDB supports many of the common operators including:
    - Arithmetic and bitwise operations
    - AND and OR logic
    - Equality and range comparisons
    - String concatenation

- The query language also supports a host of built-in functions.

2. DocumentDB SQL – SELECT Clause

The Azure portal has a Query Explorer that lets us run any SQL query against our DocumentDB database. We will use the Query Explorer to demonstrate the many different capabilities and features of the query language starting with the simplest possible query.

Step 1: Open the Azure Portal, and in the database blade, click the Query Explorer blade.



Status: Online
Subscription Id: 973ad1fe-282d-4646-898e-2e76cce75d59
Location: East Asia

Databases: myfirstdb

Collections: Families

```
SELECT * FROM c
```

Remember that queries run within the scope of a collection, and so the Query Explorer lets us choose the collection in this dropdown. We will leave it set to our Families collection that contains the three documents. Let's consider these three documents in this example.

Following is the **AndersenFamily** document.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
             { "firstName": "Thomas", "relationship":  "father" },
             { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
             {
                    "firstName": "Henriette Thaulow",
                    "gender": "female",
                    "grade": 5,
                    "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
             }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

Following is the **SmithFamily** document.

```
{
      "id": "SmithFamily",
      "parents": [
             { "familyName": "Smith", "givenName": "James" },
             { "familyName": "Curtis", "givenName": "Helen" }
      ],
      "children": [
             {
                    "givenName": "Michelle",
                    "gender": "female",
                    "grade": 1
             },
             {
                    "givenName": "John",
                    "gender": "male",
                    "grade": 7,
                    "pets": [
                           { "givenName": "Tweetie", "type": "Bird" }
                    ]
             }
      ],
      "location": {
             "state": "NY",
             "county": "Queens",
             "city": "Forest Hills"
      },
      "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

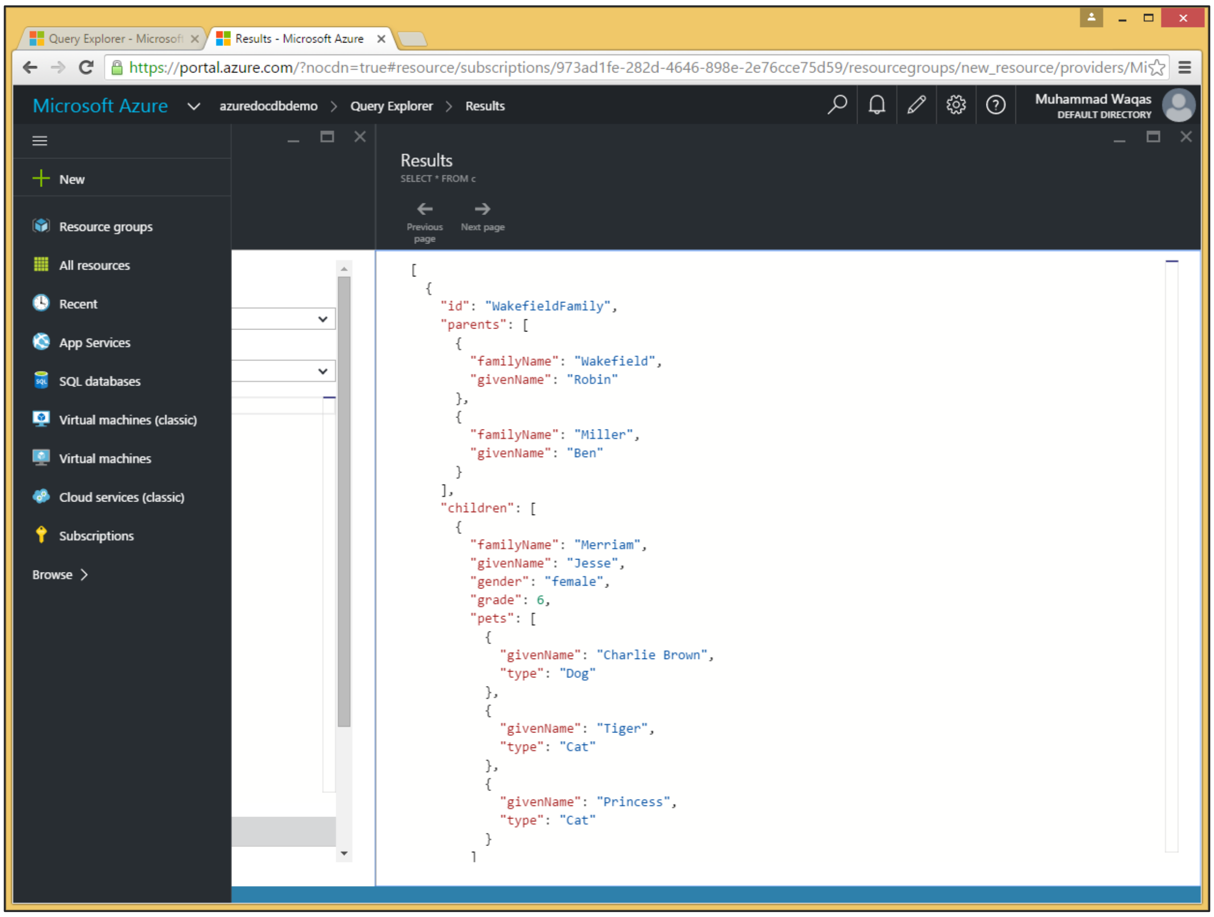
```
{
     "id": "WakefieldFamily",
     "parents": [
            { "familyName": "Wakefield", "givenName": "Robin" },
            { "familyName": "Miller", "givenName": "Ben" }
     ],
     "children": [
            {
                  "familyName": "Merriam",
                  "givenName": "Jesse",
                  "gender": "female",
                  "grade": 6,
                  "pets": [
                        { "givenName": "Charlie Brown", "type": "Dog" },
                { "givenName": "Tiger", "type": "Cat" },
                { "givenName": "Princess", "type": "Cat" }
                  ]
            },
            {
                  "familyName": "Miller",
                  "givenName": "Lisa",
                  "gender": "female",
                  "grade": 3,
                  "pets": [
                        { "givenName": "Jake", "type": "Snake" }
                  ]
            }
     ],
     "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
     "isRegistered": false}
```

The Query Explorer opens up with this simple query SELECT * FROM c, which simply retrieves all documents from the collection. Although it is simple, it's still quite different than the equivalent query in a relational database.

Step 2: In relational databases, SELECT * means return all columns while in DocumentDB. It means that you want each document in your result to be returned exactly as it's stored in the database.

But when you select specific properties and expressions instead of simply issuing a SELECT *, then you are projecting a new shape that you want for each document in the result.

Step 3: Click 'Run' to execute query and open the Results blade.



```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
```

As can be seen the WakefieldFamily, the SmithFamily, and the AndersonFamily are retrieved.

Following are the three documents which are retrieved as a result of the **SELECT * FROM c** query.

```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
      },
      {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
        "grade": 3,
        "pets": [
          {
            "givenName": "Jake",
            "type": "Snake"
          }
        ]
      }
    ],
    "location": {
      "state": "NY",
      "county": "Manhattan",
      "city": "NY"
    },
    "isRegistered": false,
    "_rid": "Ic8LAJFujgECAAAAAAAAAA==",
    "_ts": 1450541623,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgECAAAAAAAAAA==/",
    "_etag": "\"00000500-0000-0000-0000-567582370000\"",
    "_attachments": "attachments/"
  },
  {
    "id": "SmithFamily",
    "parents": [
      {
        "familyName": "Smith",
        "givenName": "James"
      },
      {
        "familyName": "Curtis",
        "givenName": "Helen"
      }
    ],
    "children": [
      {
        "givenName": "Michelle",
        "gender": "female",
        "grade": 1
      },
      {
        "givenName": "John",
        "gender": "male",
        "grade": 7,
        "pets": [
          {
            "givenName": "Tweetie",
            "type": "Bird"
          }
        ]
      }
    ],
    "location": {
      "state": "NY",
      "county": "Queens",
      "city": "Forest Hills"
    },
    "isRegistered": true,
    "_rid": "Ic8LAJFujgEDAAAAAAAAAA==",
    "_ts": 1450541623,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgEDAAAAAAAAAA==/",
    "_etag": "\"00000600-0000-0000-0000-567582370000\"",
    "_attachments": "attachments/"
  },
  {
    "id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
      {
        "firstName": "Thomas",
        "relationship": "father"
      },
      {
        "firstName": "Mary Kay",
        "relationship": "mother"
      }
    ],
    "children": [
      {
        "firstName": "Henriette Thaulow",
        "gender": "female",
        "grade": 5,
        "pets": [
          {
            "givenName": "Fluffy",
            "type": "Rabbit"
          }
        ]
      }
    ],
    "location": {
      "state": "WA",
      "county": "King",
      "city": "Seattle"
    },
    "isRegistered": true,
    "_rid": "Ic8LAJFujgEEAAAAAAAAAA==",
    "_ts": 1450541624,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgEEAAAAAAAAAA==/",
    "_etag": "\"00000700-0000-0000-0000-567582380000\"",
    "_attachments": "attachments/"
  }
]
```

However, these results also include the system-generated properties that are all prefixed with the underscore character.

3. DocumentDB SQL – FROM Clause

In this chapter, we will cover the FROM clause, which works nothing like a standard FROM clause in regular SQL.

Queries always run within the context of a specific collection and cannot join across documents within the collection, which makes us wonder why we need a FROM clause. In fact, we don't, but if we don't include it, then we won't be querying documents in the collection.

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. The FROM <from_specification> clause is optional unless the source is filtered or projected later in the query.

Let's take a look at the same example again. Following is the **AndersenFamily** document.

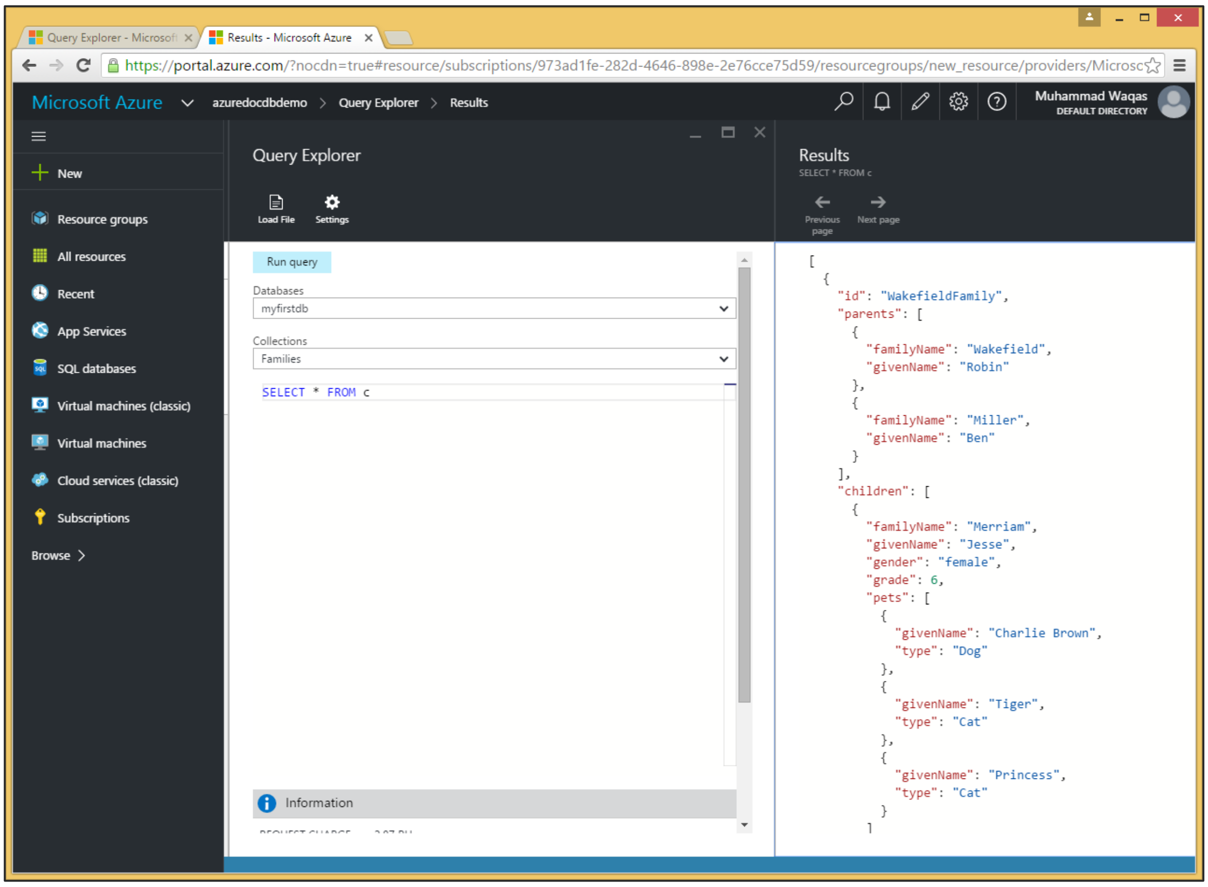
```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
             { "firstName": "Thomas", "relationship":  "father" },
             { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
             {
                    "firstName": "Henriette Thaulow",
                    "gender": "female",
                    "grade": 5,
                    "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
             }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

Following is the **SmithFamily** document.

```
{
      "id": "SmithFamily",
      "parents": [
            { "familyName": "Smith", "givenName": "James" },
            { "familyName": "Curtis", "givenName": "Helen" }
      ],
      "children": [
            {
                  "givenName": "Michelle",
                  "gender": "female",
                  "grade": 1
            },
            {
                  "givenName": "John",
                  "gender": "male",
                  "grade": 7,
                  "pets": [
                        { "givenName": "Tweetie", "type": "Bird" }
                  ]
            }
      ],
      "location": {
            "state": "NY",
            "county": "Queens",
            "city": "Forest Hills"
      },
      "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
      "id": "WakefieldFamily",
      "parents": [
            { "familyName": "Wakefield", "givenName": "Robin" },
            { "familyName": "Miller", "givenName": "Ben" }
      ],
      "children": [
            {
                  "familyName": "Merriam",
                  "givenName": "Jesse",
                  "gender": "female",
                  "grade": 6,
                  "pets": [
                        { "givenName": "Charlie Brown", "type": "Dog" },
                { "givenName": "Tiger", "type": "Cat" },
                { "givenName": "Princess", "type": "Cat" }
                  ]
            },
            {
                  "familyName": "Miller",
                  "givenName": "Lisa",
                  "gender": "female",
                  "grade": 3,
                  "pets": [
                        { "givenName": "Jake", "type": "Snake" }
                  ]
            }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```



Query Explorer

Databases: myfirstdb

Collections: Families

```
SELECT * FROM c
```

Results

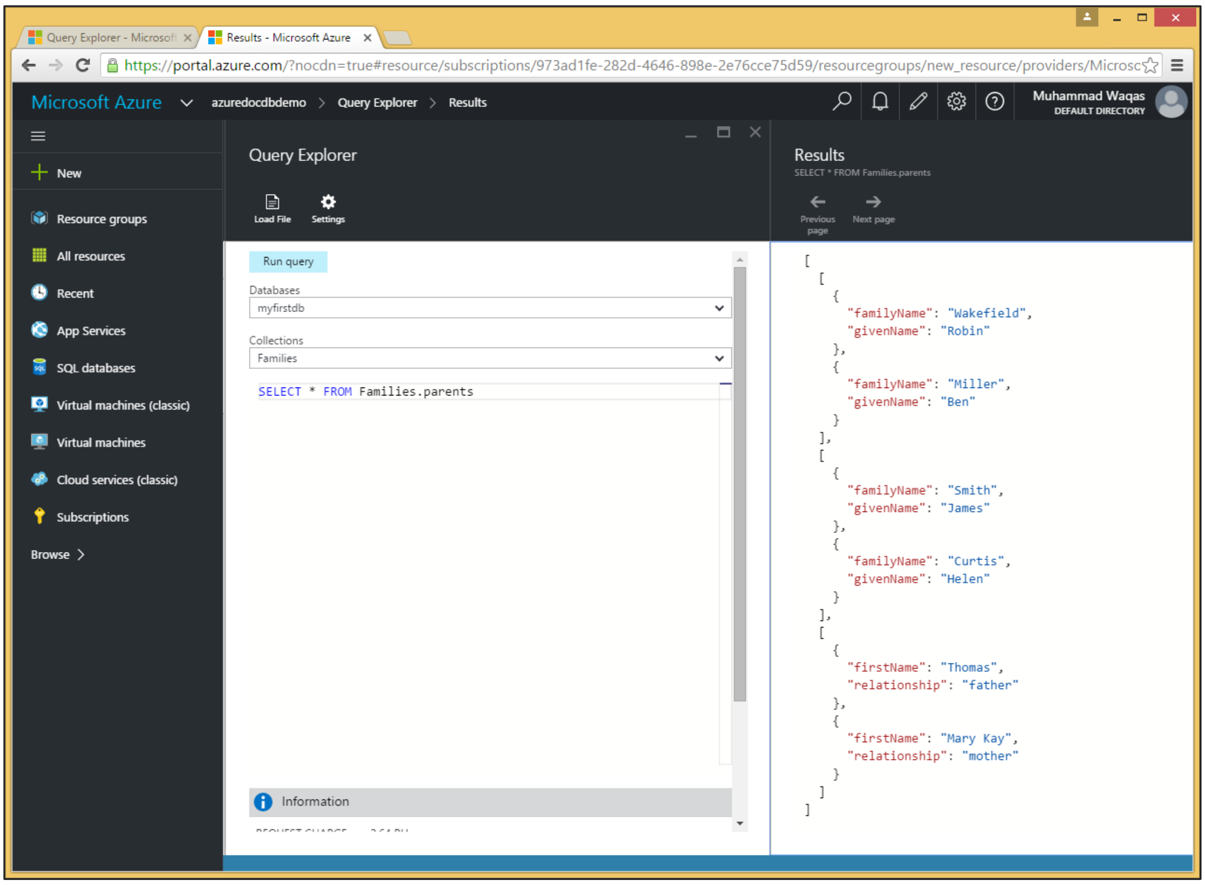
SELECT * FROM c

```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
```

In the above query, "**SELECT * FROM c**" indicates that the entire Families collection is the source over which to enumerate.

Sub-documents

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


Query Explorer - Microsoft
Results - Microsoft Azure
https://portal.azure.com/?nocdn=true#resource/subscriptions/973ad1fe-282d-4646-898e-2e76cce75d59/resourcegroups/new_resource/providers/Microsc
Microsoft Azure
azuredocdbdemo > Query Explorer > Results
Muhammad Waqas
DEFAULT DIRECTORY
New
Resource groups
All resources
Recent
App Services
SQL databases
Virtual machines (classic)
Virtual machines
Cloud services (classic)
Subscriptions
Browse
Query Explorer
Load File
Settings
Run query
Databases
myfirstdb
Collections
Families
Information
Results
SELECT * FROM Families.parents
Previous page
Next page

```
SELECT * FROM Families.parents
```

```
[
  [
    {
      "familyName": "Wakefield",
      "givenName": "Robin"
    },
    {
      "familyName": "Miller",
      "givenName": "Ben"
    }
  ],
  [
    {
      "familyName": "Smith",
      "givenName": "James"
    },
    {
      "familyName": "Curtis",
      "givenName": "Helen"
    }
  ],
  [
    {
      "firstName": "Thomas",
      "relationship": "father"
    },
    {
      "firstName": "Mary Kay",
      "relationship": "mother"
    }
  ]
]
```

When we run the following query:

```
SELECT * FROM Families.parents
```

The following sub-documents will be retrieved.

```
[
  [
    {
      "familyName": "Wakefield",
      "givenName": "Robin"
    },
    {
      "familyName": "Miller",
      "givenName": "Ben"
    }
  ],
  [
    {
      "familyName": "Smith",
      "givenName": "James"
    },
    {
      "familyName": "Curtis",
      "givenName": "Helen"
    }
  ],
  [
    {
      "firstName": "Thomas",
      "relationship": "father"
    },
    {
      "firstName": "Mary Kay",
      "relationship": "mother"
    }
  ]
]
```

As a result of this query, we can see that only the parents sub-documents are retrieved.

4. DocumentDB SQL – WHERE Clause

In this chapter, we will cover the WHERE clause, which is also optional like FROM clause. It is used to specify a condition while fetching the data in the form of JSON documents provided by the source. Any JSON document must evaluate the specified conditions to be "true" to be considered for the result. If the given condition is satisfied, only then it returns specific data in the form of JSON document(s). We can use WHERE clause to filter the records and fetch only necessary records.

We will consider the same three documents in this example. Following is the **AndersenFamily** document.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
             { "firstName": "Thomas", "relationship":  "father" },
             { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
             {
                    "firstName": "Henriette Thaulow",
                    "gender": "female",
                    "grade": 5,
                    "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
             }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

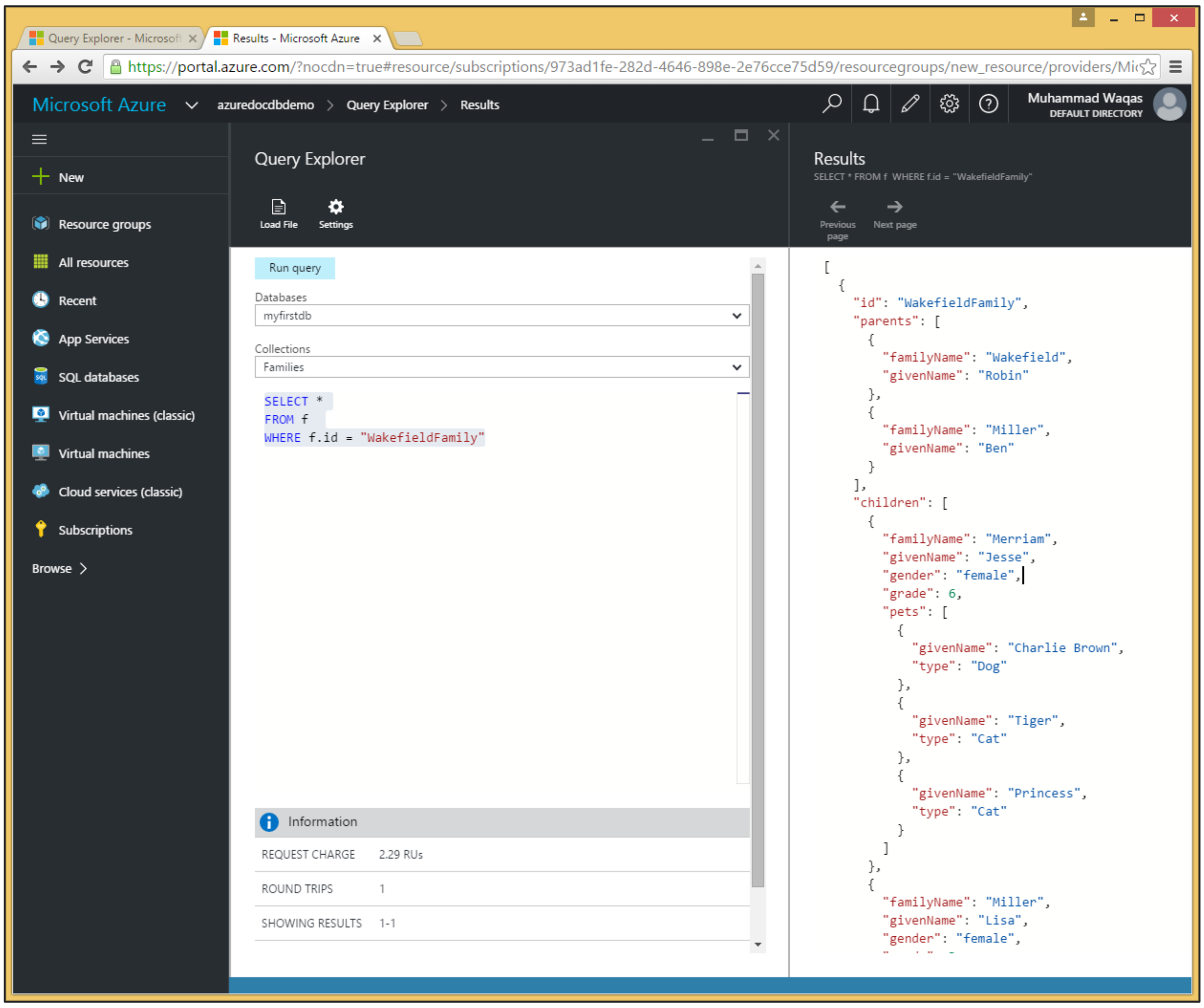
Following is the **SmithFamily** document.

```
{
      "id": "SmithFamily",
      "parents": [
             { "familyName": "Smith", "givenName": "James" },
             { "familyName": "Curtis", "givenName": "Helen" }
      ],
    "children": [
        {
            "givenName": "Michelle",
            "gender": "female",
            "grade": 1
        },
        {
            "givenName": "John",
            "gender": "male",
            "grade": 7,
            "pets": [
                { "givenName": "Tweetie", "type": "Bird" }
            ]
        }
    ],
    "location": {
        "state": "NY",
        "county": "Queens",
        "city": "Forest Hills"
    },
    "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
    "id": "WakefieldFamily",
    "parents": [
        { "familyName": "Wakefield", "givenName": "Robin" },
        { "familyName": "Miller", "givenName": "Ben" }
    ],
    "children": [
        {
            "familyName": "Merriam",
            "givenName": "Jesse",
            "gender": "female",
            "grade": 6,
            "pets": [
                  { "givenName": "Charlie Brown", "type": "Dog" },
          { "givenName": "Tiger", "type": "Cat" },
          { "givenName": "Princess", "type": "Cat" }
            ]
        },
        {
            "familyName": "Miller",
            "givenName": "Lisa",
            "gender": "female",
            "grade": 3,
            "pets": [
                  { "givenName": "Jake", "type": "Snake" }
            ]
        }
    ],
    "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
    "isRegistered": false
}
```

Let’s take a look at a simple example in which WHERE clause is used.



Databases: myfirstdb

Collections: Families

```
SELECT *
FROM f
WHERE f.id = "WakefieldFamily"
```

Information

REQUEST CHARGE	2.29 RUs
ROUND TRIPS	1
SHOWING RESULTS	1-1

Results

SELECT * FROM f WHERE f.id = "WakefieldFamily"

```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
      },
      {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
```

In this query, in WHERE clause, the (WHERE f.id = "WakefieldFamily") condition is specified.

```
SELECT *
FROM f
WHERE f.id = "WakefieldFamily"
```

When the above query is executed, it will return the complete JSON document for WakefieldFamily as shown in the following output.

```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
      },
      {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
        "grade": 3,
        "pets": [
          {
            "givenName": "Jake",
            "type": "Snake"
          }
        ]
      }
    ],
    "location": {
      "state": "NY",
      "county": "Manhattan",
      "city": "NY"
    },
    "isRegistered": false,
    "_rid": "Ic8LAJFujgECAAAAAAAAAA==",
    "_ts": 1450541623,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgECAAAAAAAAAA==/",
    "_etag": "\"00000500-0000-0000-0000-567582370000\"",
    "_attachments": "attachments/"
  }
]
```

5. DocumentDB SQL – Operators

An operator is a reserved word or a character used primarily in an SQL WHERE clause to perform operation(s), such as comparisons and arithmetic operations. DocumentDB SQL also supports a variety of scalar expressions. The most commonly used are **binary and unary expressions**.

The following SQL operators are currently supported and can be used in queries.

SQL Comparison Operators

Following is a list of all the comparison operators available in DocumentDB SQL grammar.

Operator	Description
=	Checks if the values of two operands are equal or not. If yes, then condition becomes true.
!=	Checks if the values of two operands are equal or not. If values are not equal then condition becomes true.
<>	Checks if the values of two operands are equal or not. If values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then condition becomes true.

SQL Logical Operators

Following is a list of all the logical operators available in DocumentDB SQL grammar.

Operator	Description
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. For example, NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator.

SQL Arithmetic Operators

Following is a list of all the arithmetic operators available in DocumentDB SQL grammar.

Operator	Description
+	**Addition** - Adds values on either side of the operator.
-	**Subtraction** - Subtracts the right hand operand from the left hand operand.
*	**Multiplication** - Multiplies values on either side of the operator.
/	**Division** - Divides the left hand operand by the right hand operand.
%	**Modulus** - Divides the left hand operand by the right hand operand and returns the remainder.

We will consider the same documents in this example as well. Following is the **AndersenFamily** document.

```
{
     "id": "AndersenFamily",
     "lastName": "Andersen",
     "parents": [
           { "firstName": "Thomas", "relationship":  "father" },
           { "firstName": "Mary Kay", "relationship":  "mother" }
     ],
     "children": [
           {
                 "firstName": "Henriette Thaulow",
                 "gender": "female",
                 "grade": 5,
                 "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
           }
     ],
     "location": { "state": "WA", "county": "King", "city": "Seattle" },
     "isRegistered": true
}
```

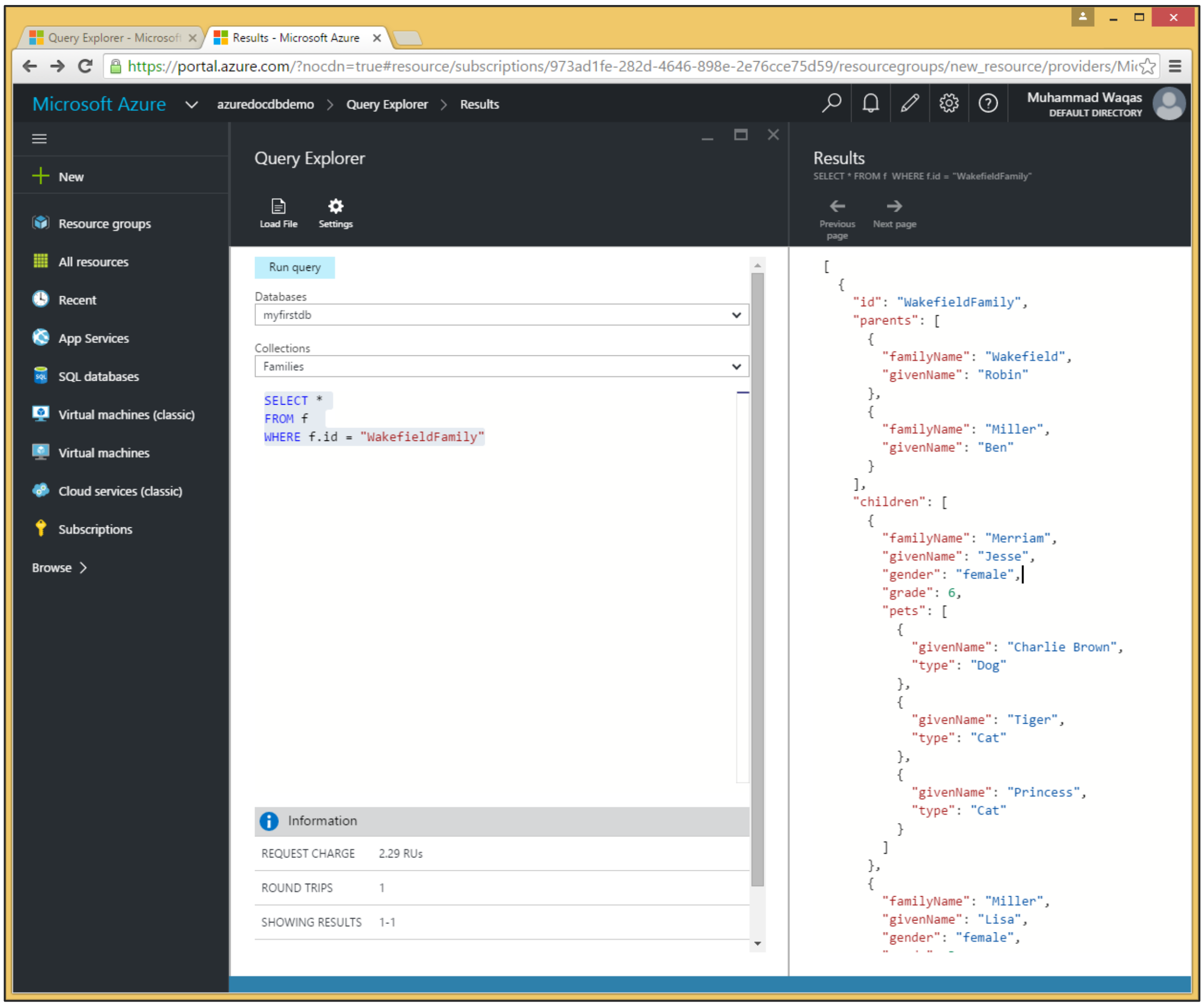
Following is the **SmithFamily** document.

```
{
     "id": "SmithFamily",
     "parents": [
           { "familyName": "Smith", "givenName": "James" },
           { "familyName": "Curtis", "givenName": "Helen" }
     ],
     "children": [
           {
                 "givenName": "Michelle",
                 "gender": "female",
                 "grade": 1
           },
           {
                 "givenName": "John",
                 "gender": "male",
                "grade": 7,
                "pets": [
                        { "givenName": "Tweetie", "type": "Bird" }
                ]
        }
    ],
    "location": {
        "state": "NY",
        "county": "Queens",
        "city": "Forest Hills"
    },
    "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
    "id": "WakefieldFamily",
    "parents": [
        { "familyName": "Wakefield", "givenName": "Robin" },
        { "familyName": "Miller", "givenName": "Ben" }
    ],
    "children": [
        {
                "familyName": "Merriam",
                "givenName": "Jesse",
                "gender": "female",
                "grade": 6,
                "pets": [
                        { "givenName": "Charlie Brown", "type": "Dog" },
              { "givenName": "Tiger", "type": "Cat" },
              { "givenName": "Princess", "type": "Cat" }
                ]
        },
        {
                "familyName": "Miller",
                "givenName": "Lisa",
                "gender": "female",
                "grade": 3,
                    "pets": [
                            { "givenName": "Jake", "type": "Snake" }
                    ]
            }
    ],
    "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
    "isRegistered": false
}
```

Let’s take a look at a simple example in which a comparison operator is used in WHERE clause.



Query Explorer

Databases: myfirstdb

Collections: Families

```
SELECT *
FROM f
WHERE f.id = "WakefieldFamily"
```

REQUEST CHARGE 2.29 RUs
ROUND TRIPS 1
SHOWING RESULTS 1-1

Results

SELECT * FROM f WHERE f.id = "WakefieldFamily"

```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
      },
      {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
```

In this query, in WHERE clause, the (WHERE f.id = "WakefieldFamily") condition is specified, and it will retrieve the document whose id is equal to WakefieldFamily.

```
SELECT *
FROM f
WHERE f.id = "WakefieldFamily"
```

When the above query is executed, it will return the complete JSON document for WakefieldFamily as shown in the following output.

```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
      },
      {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
        "grade": 3,
        "pets": [
          {
            "givenName": "Jake",
            "type": "Snake"
          }
        ]
      }
    ],
    "location": {
      "state": "NY",
      "county": "Manhattan",
      "city": "NY"
    },
    "isRegistered": false,
    "_rid": "Ic8LAJFujgECAAAAAAAAAA==",
    "_ts": 1450541623,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgECAAAAAAAAAA==/",
    "_etag": "\"00000500-0000-0000-0000-567582370000\"",
    "_attachments": "attachments/"
  }
]
```

Let's take a look at another example in which the query will retrieve the children data whose grade is greater than 5.

```
SELECT *
FROM Families.children[0] c
WHERE (c.grade > 5)
```

When the above query is executed, it will retrieve the following sub document as shown in the output.

```
[
  {
    "familyName": "Merriam",
    "givenName": "Jesse",
    "gender": "female",
    "grade": 6,
    "pets": [
      {
        "givenName": "Charlie Brown",
        "type": "Dog"
      },
      {
        "givenName": "Tiger",
        "type": "Cat"
      },
      {
        "givenName": "Princess",
        "type": "Cat"
      }
    ]
  }
]
```

6. DocumentDB SQL – BETWEEN Keyword

The BETWEEN keyword is used to express queries against ranges of values like in SQL. BETWEEN can be used against strings or numbers. 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, in some document it is possible that you might have "grade" as a number and in other documents it might be strings. In these cases, a comparison between two different types of results is "undefined", and the document will be skipped.

Let us consider the three documents from the previous example. Following is the **AndersenFamily** document.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
             { "firstName": "Thomas", "relationship":  "father" },
             { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
             {
                    "firstName": "Henriette Thaulow",
                    "gender": "female",
                    "grade": 5,
                    "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
             }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

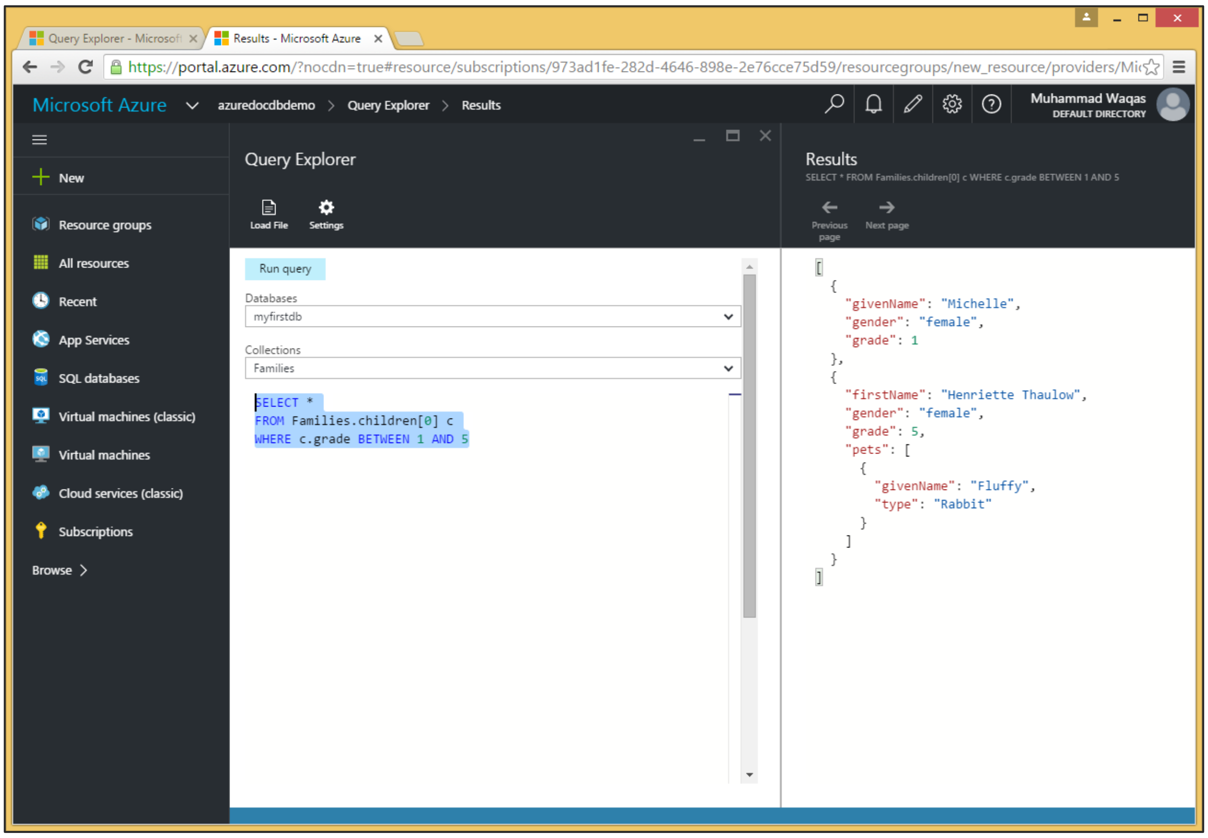
Following is the **SmithFamily** document.

```
{
      "id": "SmithFamily",
      "parents": [
             { "familyName": "Smith", "givenName": "James" },
             { "familyName": "Curtis", "givenName": "Helen" }
      ],
      "children": [
            {
                  "givenName": "Michelle",
                  "gender": "female",
                  "grade": 1
            },
            {
                  "givenName": "John",
                  "gender": "male",
                  "grade": 7,
                  "pets": [
                        { "givenName": "Tweetie", "type": "Bird" }
                  ]
            }
      ],
      "location": {
            "state": "NY",
            "county": "Queens",
            "city": "Forest Hills"
      },
      "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
      "id": "WakefieldFamily",
      "parents": [
            { "familyName": "Wakefield", "givenName": "Robin" },
            { "familyName": "Miller", "givenName": "Ben" }
      ],
      "children": [
            {
                  "familyName": "Merriam",
                  "givenName": "Jesse",
                  "gender": "female",
                  "grade": 6,
                  "pets": [
                        { "givenName": "Charlie Brown", "type": "Dog" },
                { "givenName": "Tiger", "type": "Cat" },
                { "givenName": "Princess", "type": "Cat" }
                  ]
            },
            {
                  "familyName": "Miller",
                  "givenName": "Lisa",
                  "gender": "female",
                  "grade": 3,
                  "pets": [
                        { "givenName": "Jake", "type": "Snake" }
                  ]
            }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```

Let's take a look at an example, where the query returns all family documents in which the first child's grade is between 1-5 (both inclusive).


Query Explorer - Microsoft
Results - Microsoft Azure
https://portal.azure.com/?nocdn=true#resource/subscriptions/973ad1fe-282d-4646-898e-2e76cce75d59/resourcegroups/new_resource/providers/Mi
Microsoft Azure
azuredocdbdemo > Query Explorer > Results
Muhammad Waqas
DEFAULT DIRECTORY
New
Resource groups
All resources
Recent
App Services
SQL databases
Virtual machines (classic)
Virtual machines
Cloud services (classic)
Subscriptions
Browse >
Query Explorer
Load File
Settings
Run query
Results
Previous page
Next page

Databases: myfirstdb

Collections: Families

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

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

```
[
  {
    "givenName": "Michelle",
    "gender": "female",
    "grade": 1
  },
  {
    "firstName": "Henriette Thaulow",
    "gender": "female",
    "grade": 5,
    "pets": [
      {
        "givenName": "Fluffy",
        "type": "Rabbit"
      }
    ]
  }
]
```

Following is the query in which BETWEEN keyword is used and then AND logical operator.

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

When the above query is executed, it produces the following output.

```
[
  {
    "givenName": "Michelle",
    "gender": "female",
    "grade": 1
  },
  {
    "firstName": "Henriette Thaulow",
    "gender": "female",
    "grade": 5,
    "pets": [
      {
        "givenName": "Fluffy",
        "type": "Rabbit"
      }
    ]
  }
]
```

To display the grades outside the range of the previous example, use NOT BETWEEN as shown in the following query.

```
SELECT *
FROM Families.children[0] c
WHERE c.grade NOT BETWEEN 1 AND 5
```

When this query is executed. It produces the following output.

```
[
  {
    "familyName": "Merriam",
    "givenName": "Jesse",
    "gender": "female",
    "grade": 6,
    "pets": [
      {
        "givenName": "Charlie Brown",
        "type": "Dog"
      },
      {
        "givenName": "Tiger",
        "type": "Cat"
      },
      {        "givenName": "Princess",
        "type": "Cat"
      }
    ]
  }]
```

7. DocumentDB SQL – IN Keyword

The IN keyword can be used to check whether a specified value matches any value in a list. The IN operator allows you to specify multiple values in a WHERE clause. IN is equivalent to chaining multiple OR clauses.

The similar three documents are considered as done in earlier examples. Following is the **AndersenFamily** document.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
             { "firstName": "Thomas", "relationship":  "father" },
             { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
             {
                    "firstName": "Henriette Thaulow",
                    "gender": "female",
                    "grade": 5,
                    "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
             }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

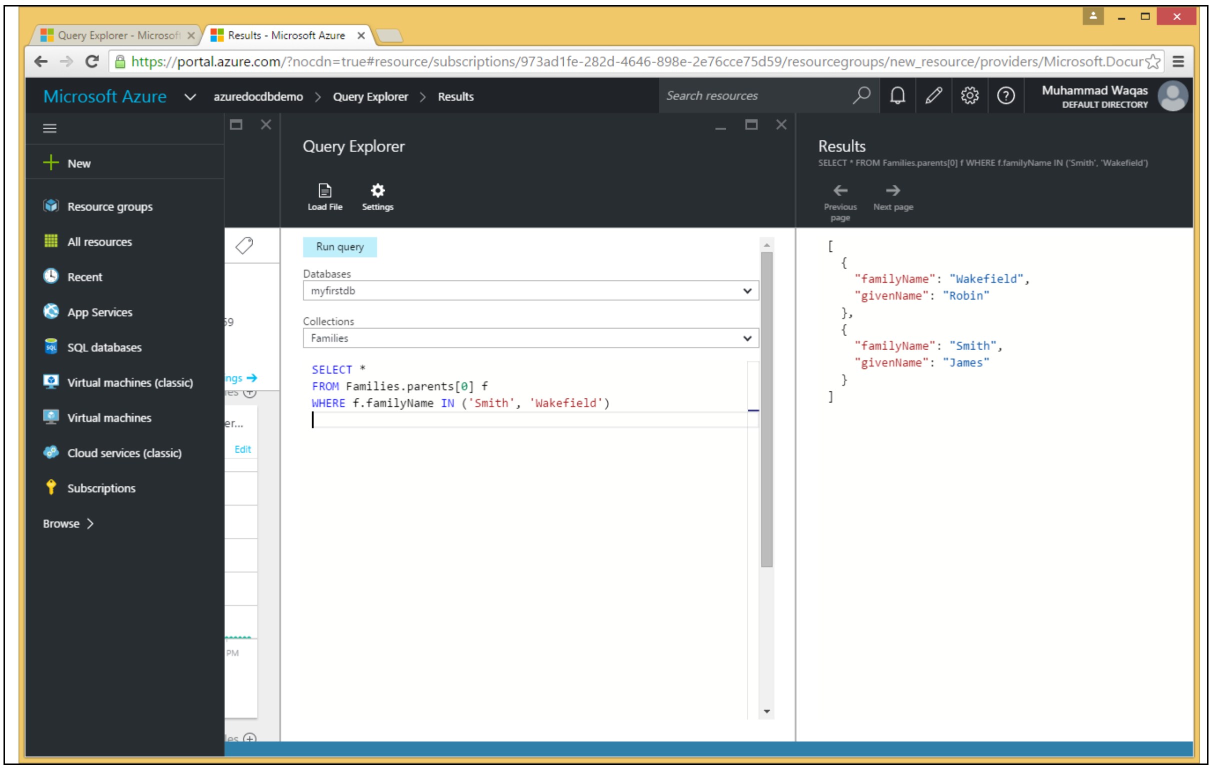
Following is the **SmithFamily** document.

```
{
      "id": "SmithFamily",
      "parents": [
             { "familyName": "Smith", "givenName": "James" },
             { "familyName": "Curtis", "givenName": "Helen" }
      ],
      "children": [
             {
                "givenName": "Michelle",
                "gender": "female",
                "grade": 1
            },
            {
                "givenName": "John",
                "gender": "male",
                "grade": 7,
                "pets": [
                    { "givenName": "Tweetie", "type": "Bird" }
                ]
            }
        ],
        "location": {
            "state": "NY",
            "county": "Queens",
            "city": "Forest Hills"
        },
        "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
        "id": "WakefieldFamily",
        "parents": [
            { "familyName": "Wakefield", "givenName": "Robin" },
            { "familyName": "Miller", "givenName": "Ben" }
        ],
        "children": [
            {
                "familyName": "Merriam",
                "givenName": "Jesse",
                "gender": "female",
                "grade": 6,
                "pets": [
                    { "givenName": "Charlie Brown", "type": "Dog" },
                { "givenName": "Tiger", "type": "Cat" },
                        { "givenName": "Princess", "type": "Cat" }
                    ]
            },
            {
                    "familyName": "Miller",
                    "givenName": "Lisa",
                    "gender": "female",
                    "grade": 3,
                    "pets": [
                            { "givenName": "Jake", "type": "Snake" }
                    ]
            }
    ],
    "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
    "isRegistered": false
}
```

Let’s take a look at a simple example.



Databases: myfirstdb

Collections: Families

```
SELECT *
FROM Families.parents[0] f
WHERE f.familyName IN ('Smith', 'Wakefield')
```

SELECT * FROM Families.parents[0] f WHERE f.familyName IN ('Smith', 'Wakefield')

```
[
  {
    "familyName": "Wakefield",
    "givenName": "Robin"
  },
  {
    "familyName": "Smith",
    "givenName": "James"
  }
]
```

Following is the query which will retrieve the data whose familyName is either “Smith” or Wakefield.

```
SELECT *
FROM Families.parents[0] f
WHERE f.familyName IN ('Smith', 'Wakefield')
```

When the above query is executed, it produces the following output.

```
[
  {
    "familyName": "Wakefield",
    "givenName": "Robin"
  },
  {
    "familyName": "Smith",
    "givenName": "James"
  }
]
```

Let’s consider another simple example in which all family documents will be retrieved where the id is one of "SmithFamily" or "AndersenFamily". Following is the query.

```
SELECT *
FROM Families
WHERE Families.id IN ('SmithFamily', 'AndersenFamily')
```

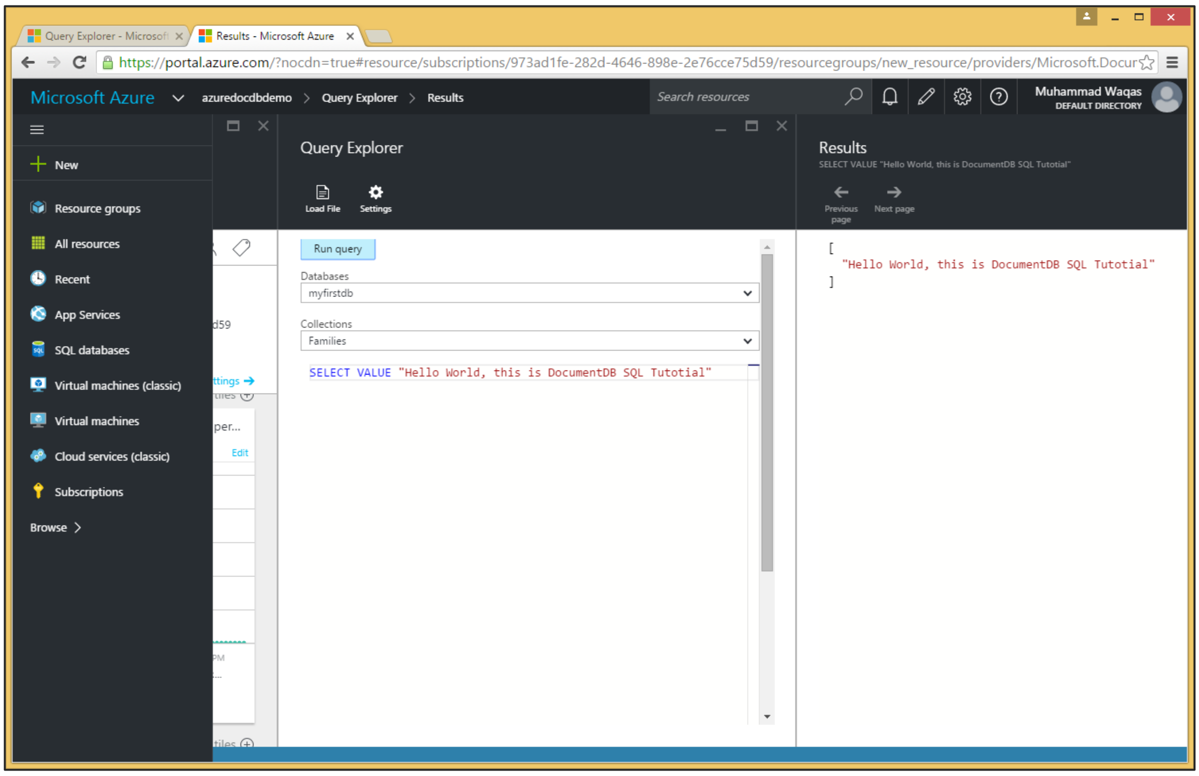
When the above query is executed, it produces the following output.

```
[
  {
    "id": "SmithFamily",
    "parents": [
      {
        "familyName": "Smith",
        "givenName": "James"
      },
      {
        "familyName": "Curtis",
        "givenName": "Helen"
      }
    ],
    "children": [
      {
        "givenName": "Michelle",
        "gender": "female",
        "grade": 1
      },
      {
        "givenName": "John",
        "gender": "male",
        "grade": 7,
        "pets": [
          {
            "givenName": "Tweetie",
            "type": "Bird"
          }
        ]
      }
    ],
    "location": {
      "state": "NY",
      "county": "Queens",
      "city": "Forest Hills"
    },
    "isRegistered": true,
    "_rid": "Ic8LAJFujgEDAAAAAAAAAA==",
    "_ts": 1450541623,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgEDAAAAAAAAAA==/",
    "_etag": "\"00000600-0000-0000-0000-567582370000\"",
    "_attachments": "attachments/"
  },
  {
    "id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
      {
        "firstName": "Thomas",
        "relationship": "father"
      },
      {
        "firstName": "Mary Kay",
        "relationship": "mother"
      }
    ],
    "children": [
      {
        "firstName": "Henriette Thaulow",
        "gender": "female",
        "grade": 5,
        "pets": [
          {
            "givenName": "Fluffy",
            "type": "Rabbit"
          }
        ]
      }
    ],
    "location": {      "state": "WA",
      "county": "King",
      "city": "Seattle"    },
    "isRegistered": true,
    "_rid": "Ic8LAJFujgEEAAAAAAAAAA==",
    "_ts": 1450541624,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgEEAAAAAAAAAA==/",
    "_etag": "\"00000700-0000-0000-0000-567582380000\"",
    "_attachments": "attachments/"
  }]
```

8. DocumentDB SQL – VALUE Keyword

When you know you're only returning a single value, then the VALUE keyword can help produce a leaner result set by avoiding the overhead of creating a full-blown object. The VALUE keyword provides a way to return JSON value.

Let's take a look at a simple example.


Query Explorer - Microsoft
Results - Microsoft Azure
https://portal.azure.com/?nocdn=true#resource/subscriptions/973ad1fe-282d-4646-898e-2e76cce75d59/resourcegroups/new_resource/providers/Microsoft.Docum
Microsoft Azure
azuredocdbdemo > Query Explorer > Results
Search resources
Muhammad Waqas
DEFAULT DIRECTORY
New
Resource groups
All resources
Recent
App Services
SQL databases
Virtual machines (classic)
Virtual machines
Cloud services (classic)
Subscriptions
Browse
Query Explorer
Load File
Settings
Run query
Results
Previous page
Next page

Databases: myfirstdb

Collections: Families

```
SELECT VALUE "Hello World, this is DocumentDB SQL Tutotial"
```

Results: SELECT VALUE "Hello World, this is DocumentDB SQL Tutotial"

```
[
  "Hello World, this is DocumentDB SQL Tutotial"
]
```

Following is the query with VALUE keyword.

```
SELECT VALUE "Hello World, this is DocumentDB SQL Tutorial"
```

When this query is executed, it returns the scalar "Hello World, this is DocumentDB SQL Tutorial".

```
[
  "Hello World, this is DocumentDB SQL Tutorial"
]
```

In another example, let’s consider the three documents from the previous examples.

Following is the **AndersenFamily** document.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
            { "firstName": "Thomas", "relationship":  "father" },
            { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
            {
                  "firstName": "Henriette Thaulow",
                  "gender": "female",
                  "grade": 5,
                  "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
            }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

Following is the **SmithFamily** document.

```
{
      "id": "SmithFamily",
      "parents": [
            { "familyName": "Smith", "givenName": "James" },
            { "familyName": "Curtis", "givenName": "Helen" }
      ],
      "children": [
            {
                  "givenName": "Michelle",
                  "gender": "female",
                  "grade": 1
            },
            {
                  "givenName": "John",
                  "gender": "male",
                  "grade": 7,
                  "pets": [
                        { "givenName": "Tweetie", "type": "Bird" }
                  ]
            }
      ],
      "location": {
            "state": "NY",
            "county": "Queens",
            "city": "Forest Hills"
      },
      "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
        "id": "WakefieldFamily",
        "parents": [
              { "familyName": "Wakefield", "givenName": "Robin" },
              { "familyName": "Miller", "givenName": "Ben" }
        ],
        "children": [
              {
                    "familyName": "Merriam",
                    "givenName": "Jesse",
                    "gender": "female",
                    "grade": 6,
                    "pets": [
                          { "givenName": "Charlie Brown", "type": "Dog" },
                  { "givenName": "Tiger", "type": "Cat" },
                  { "givenName": "Princess", "type": "Cat" }
                    ]
              },
              {
                    "familyName": "Miller",
                    "givenName": "Lisa",
                    "gender": "female",
                    "grade": 3,
                    "pets": [
                          { "givenName": "Jake", "type": "Snake" }
                    ]
              }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```

Following is the query.

```
SELECT VALUE f.location
FROM Families f
```

When this query is executed, it return the returns the address without the location label.

```
[
  {
    "state": "NY",
    "county": "Manhattan",
    "city": "NY"
  },
  {
    "state": "NY",
    "county": "Queens",
    "city": "Forest Hills"
  },
  {
    "state": "WA",
    "county": "King",
    "city": "Seattle"
  }
]
```

If we now specify the same query without VALUE Keyword, then it will return the address with location label. Following is the query.

```
SELECT f.location
FROM Families f
```

When this query is executed, it produces the following output.

```
[
  {
    "location": {
      "state": "NY",
      "county": "Manhattan",
      "city": "NY"
    }
  },
  {
    "location": {
      "state": "NY",
      "county": "Queens",
      "city": "Forest Hills"
    }
  },
  {
    "location": {
      "state": "WA",
      "county": "King",
      "city": "Seattle"
    }
  }
]
```

9. DocumentDB SQL – ORDER BY Clause

Microsoft Azure DocumentDB supports querying documents using SQL over JSON documents. You can sort documents in the collection on numbers and strings using an ORDER BY clause in your query. The clause can include an optional ASC/DESC argument to specify the order in which results must be retrieved.

We will consider the same documents as in the previous examples.

Following is the **AndersenFamily** document.

```
{
	"id": "AndersenFamily",
	"lastName": "Andersen",
	"parents": [
		{ "firstName": "Thomas", "relationship":  "father" },
		{ "firstName": "Mary Kay", "relationship":  "mother" }
	],
	"children": [
		{
			"firstName": "Henriette Thaulow",
			"gender": "female",
			"grade": 5,
			"pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
		}
	],
	"location": { "state": "WA", "county": "King", "city": "Seattle" },
	"isRegistered": true
}
```

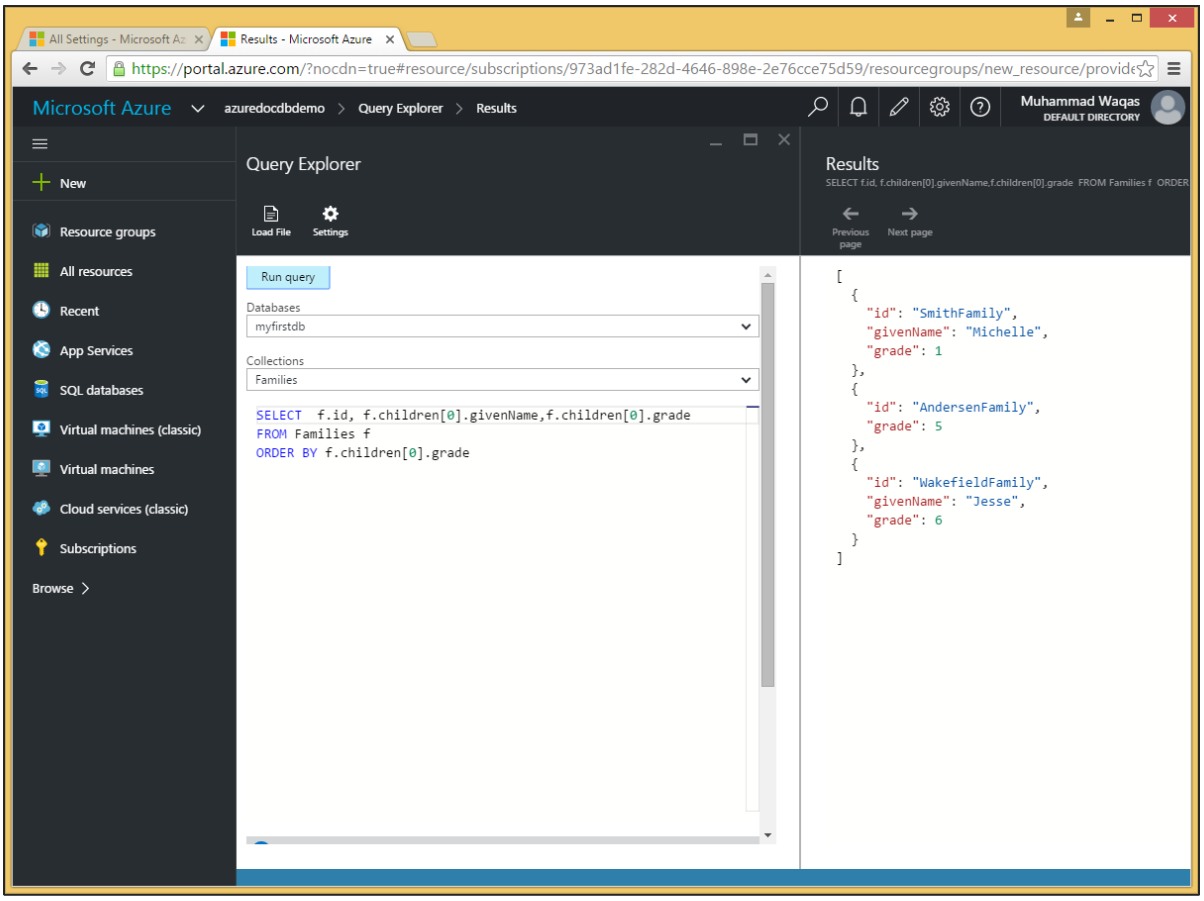
Following is the **SmithFamily** document.

```
{
	"id": "SmithFamily",
	"parents": [
		{ "familyName": "Smith", "givenName": "James" },
		{ "familyName": "Curtis", "givenName": "Helen" }
	],
	"children": [
            {
                  "givenName": "Michelle",
                  "gender": "female",
                  "grade": 1
            },
            {
                  "givenName": "John",
                  "gender": "male",
                  "grade": 7,
                  "pets": [
                        { "givenName": "Tweetie", "type": "Bird" }
                  ]
            }
      ],
      "location": {
            "state": "NY",
            "county": "Queens",
            "city": "Forest Hills"
      },
      "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
      "id": "WakefieldFamily",
      "parents": [
            { "familyName": "Wakefield", "givenName": "Robin" },
            { "familyName": "Miller", "givenName": "Ben" }
      ],
      "children": [
            {
                  "familyName": "Merriam",
                  "givenName": "Jesse",
                  "gender": "female",
                  "grade": 6,
                  "pets": [
                        { "givenName": "Charlie Brown", "type": "Dog" },
                { "givenName": "Tiger", "type": "Cat" },
                { "givenName": "Princess", "type": "Cat" }
                  ]
            },
            {
                  "familyName": "Miller",
                  "givenName": "Lisa",
                  "gender": "female",
                  "grade": 3,
                  "pets": [
                        { "givenName": "Jake", "type": "Snake" }
                  ]
            }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```

Let’s take a look at a simple example.


All Settings - Microsoft Az
Results - Microsoft Azure
https://portal.azure.com/?nocdn=true#resource/subscriptions/973ad1fe-282d-4646-898e-2e76cce75d59/resourcegroups/new_resource/provide
Microsoft Azure
azuredocdbdemo > Query Explorer > Results
Muhammad Waqas
DEFAULT DIRECTORY
New
Resource groups
All resources
Recent
App Services
SQL databases
Virtual machines (classic)
Virtual machines
Cloud services (classic)
Subscriptions
Browse
Query Explorer
Load File
Settings
Run query
Databases
myfirstdb
Collections
Families
Results
SELECT f.id, f.children[0].givenName,f.children[0].grade FROM Families f ORDER
Previous page
Next page

```
SELECT  f.id, f.children[0].givenName,f.children[0].grade
FROM Families f
ORDER BY f.children[0].grade
```

```
[
  {
    "id": "SmithFamily",
    "givenName": "Michelle",
    "grade": 1
  },
  {
    "id": "AndersenFamily",
    "grade": 5
  },
  {
    "id": "WakefieldFamily",
    "givenName": "Jesse",
    "grade": 6
  }
]
```

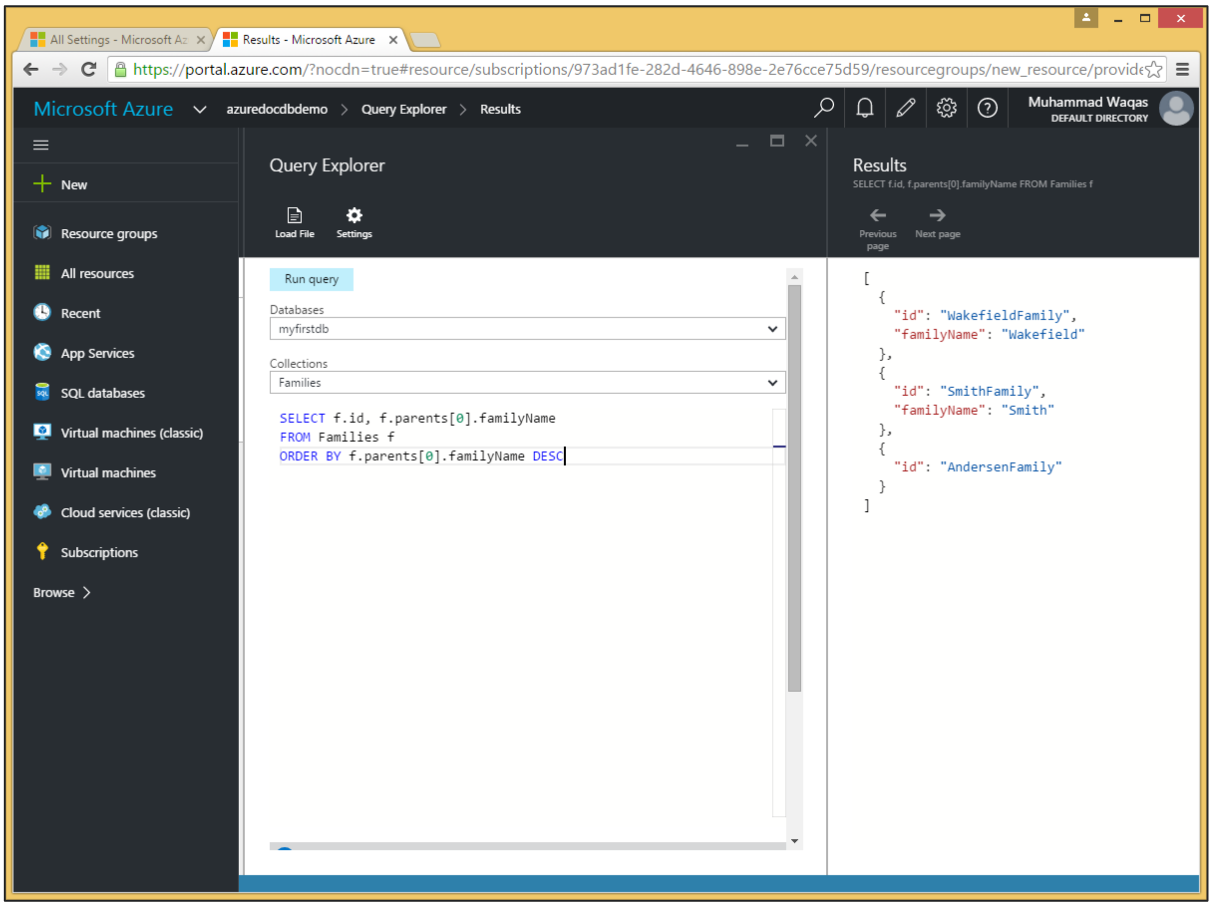
Following is the query which contains the ORDER BY keyword.

```
SELECT  f.id, f.children[0].givenName,f.children[0].grade
FROM Families f
ORDER BY f.children[0].grade
```

When the above query is executed, it produces the following output.

```
[
  {
    "id": "SmithFamily",
    "givenName": "Michelle",
    "grade": 1
  },
  {
    "id": "AndersenFamily",
    "grade": 5
  },
  {
    "id": "WakefieldFamily",
    "givenName": "Jesse",
    "grade": 6
  }
]
```

Let’s consider another simple example.



```
SELECT f.id, f.parents[0].familyName
FROM Families f
ORDER BY f.parents[0].familyName DESC
```

```
[
  {
    "id": "WakefieldFamily",
    "familyName": "Wakefield"
  },
  {
    "id": "SmithFamily",
    "familyName": "Smith"
  },
  {
    "id": "AndersenFamily"
  }
]
```

Following is the query which contains the ORDER BY keyword and DESC optional keyword.

```
SELECT f.id, f.parents[0].familyName
FROM Families f
ORDER BY f.parents[0].familyName DESC
```

When the above query is executed, it will produce the following output.

```
[
  {    "id": "WakefieldFamily",
    "familyName": "Wakefield"
  },
  {
    "id": "SmithFamily",
    "familyName": "Smith"
  },
  {    "id": "AndersenFamily"
  }]
```

10. DocumentDB SQL – Iteration

In DocumentDB SQL, Microsoft has added a new construct which can be used with IN keyword to provide support for iterating over JSON arrays. The support for iteration is provided in the FROM clause.

We will consider similar three documents from the previous examples again.

Following is the **AndersenFamily** document.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
             { "firstName": "Thomas", "relationship":  "father" },
             { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
             {
                   "firstName": "Henriette Thaulow",
                   "gender": "female",
                   "grade": 5,
                   "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
             }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

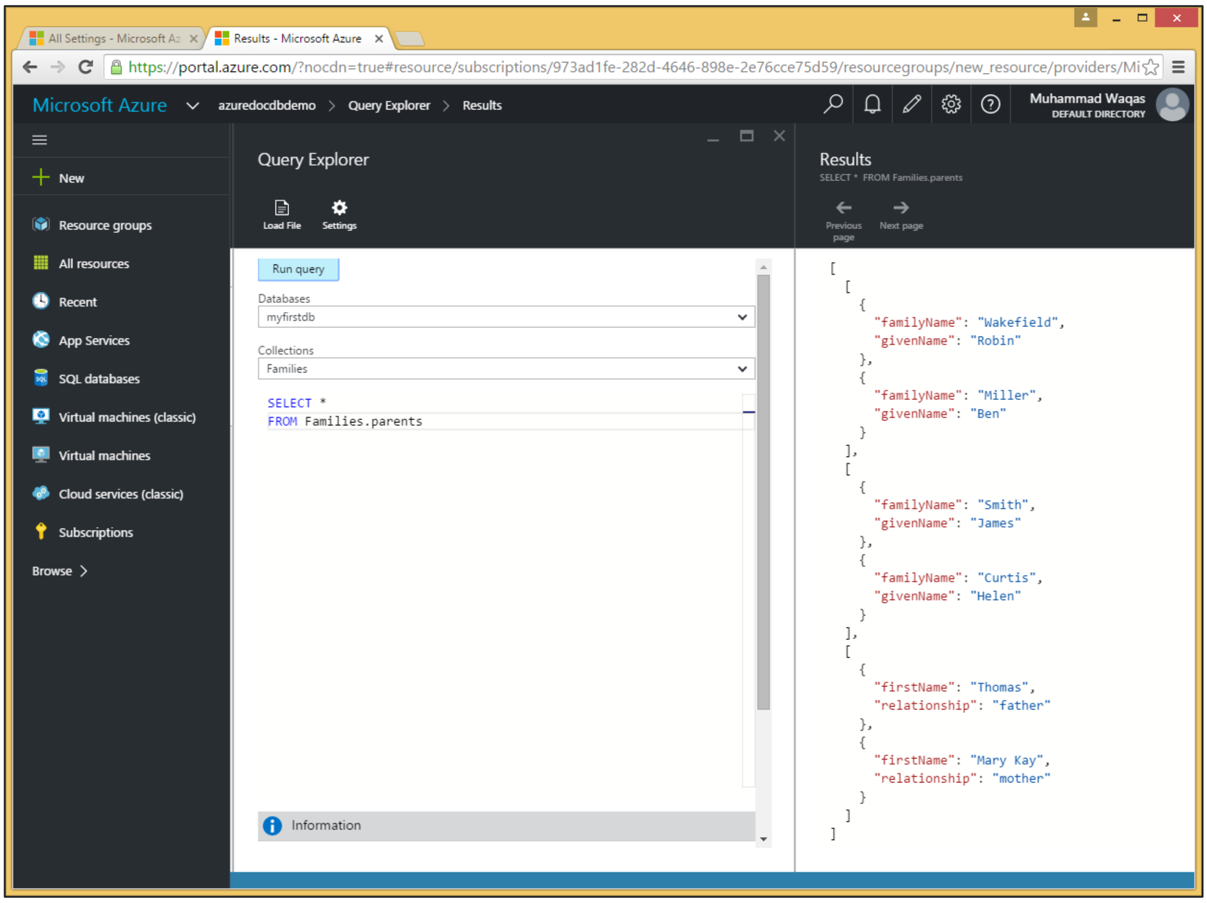
Following is the **SmithFamily** document.

```
{
      "id": "SmithFamily",
      "parents": [
             { "familyName": "Smith", "givenName": "James" },
             { "familyName": "Curtis", "givenName": "Helen" }
      ],
      "children": [
             {
                "givenName": "Michelle",
                "gender": "female",
                "grade": 1
            },
            {
                "givenName": "John",
                "gender": "male",
                "grade": 7,
                "pets": [
                    { "givenName": "Tweetie", "type": "Bird" }
                ]
            }
        ],
        "location": {
            "state": "NY",
            "county": "Queens",
            "city": "Forest Hills"
        },
        "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
        "id": "WakefieldFamily",
        "parents": [
            { "familyName": "Wakefield", "givenName": "Robin" },
            { "familyName": "Miller", "givenName": "Ben" }
        ],
        "children": [
            {
                "familyName": "Merriam",
                "givenName": "Jesse",
                "gender": "female",
                "grade": 6,
                "pets": [
                    { "givenName": "Charlie Brown", "type": "Dog" },
              { "givenName": "Tiger", "type": "Cat" },
                    { "givenName": "Princess", "type": "Cat" }
                      ]
               },
               {
                      "familyName": "Miller",
                      "givenName": "Lisa",
                      "gender": "female",
                      "grade": 3,
                      "pets": [
                             { "givenName": "Jake", "type": "Snake" }
                      ]
               }
        ],
        "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
        "isRegistered": false
}
```

Let's take a look at a simple example without IN keyword in FROM clause.



Databases: myfirstdb

Collections: Families

```
SELECT *
FROM Families.parents
```

Results

SELECT * FROM Families.parents

```
[
  [
    {
      "familyName": "Wakefield",
      "givenName": "Robin"
    },
    {
      "familyName": "Miller",
      "givenName": "Ben"
    }
  ],
  [
    {
      "familyName": "Smith",
      "givenName": "James"
    },
    {
      "familyName": "Curtis",
      "givenName": "Helen"
    }
  ],
  [
    {
      "firstName": "Thomas",
      "relationship": "father"
    },
    {
      "firstName": "Mary Kay",
      "relationship": "mother"
    }
  ]
]
```

Following is the query which will return all the parents from the Families collection.

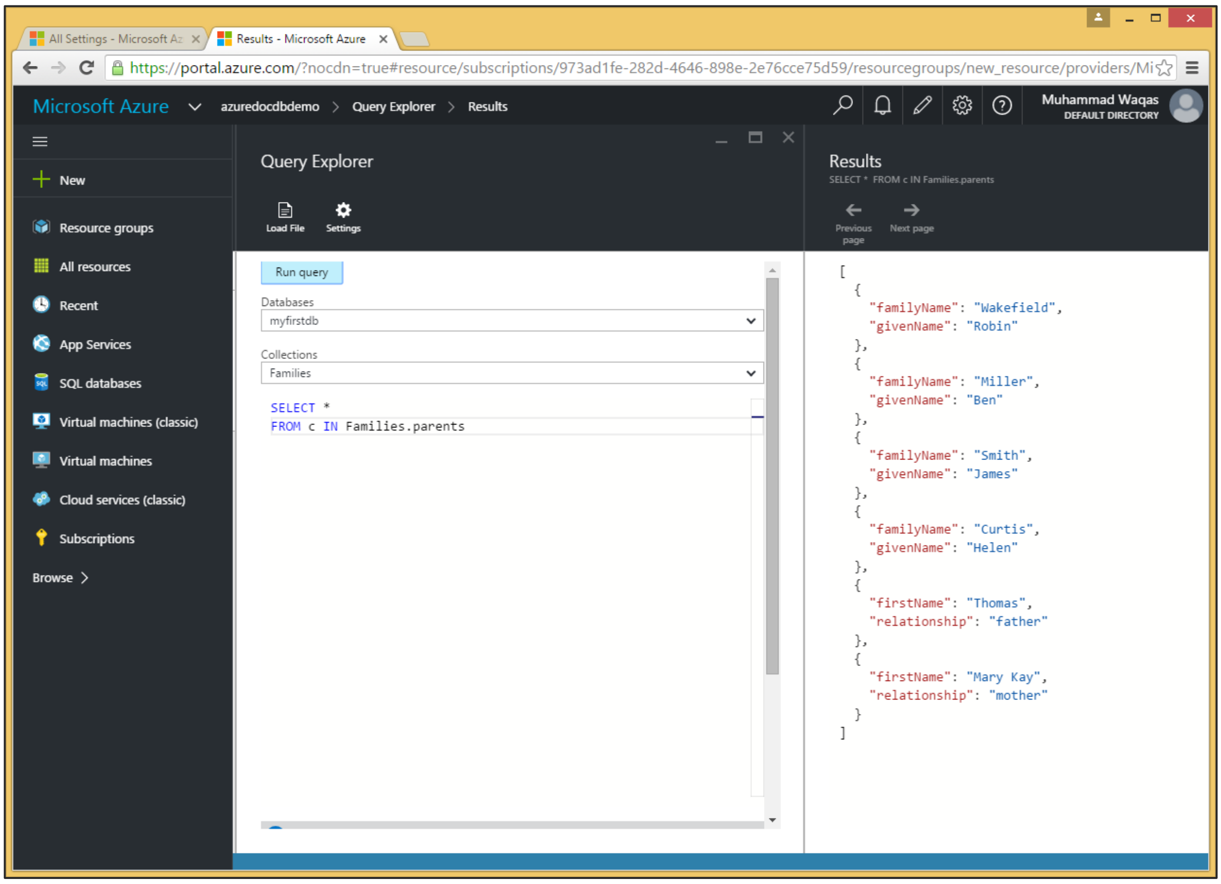
```
SELECT *
FROM Families.parents
```

When the above query is executed, it produces the following output.

```
[
  [
    {
      "familyName": "Wakefield",
      "givenName": "Robin"
    },
    {
      "familyName": "Miller",
      "givenName": "Ben"
    }
  ],
  [
    {
      "familyName": "Smith",
      "givenName": "James"
    },
    {
      "familyName": "Curtis",
      "givenName": "Helen"
    }
  ],
  [
    {
      "firstName": "Thomas",
      "relationship": "father"
    },
    {      "firstName": "Mary Kay",
      "relationship": "mother"
    }
  ]
]
```

As can be seen in the above output, the parents of each family is displayed in a separate JSON array.

Let's take a look at the same example, however this time we will use the IN keyword in FROM clause.



```
SELECT *
FROM c IN Families.parents
```

```
[
  {
    "familyName": "Wakefield",
    "givenName": "Robin"
  },
  {
    "familyName": "Miller",
    "givenName": "Ben"
  },
  {
    "familyName": "Smith",
    "givenName": "James"
  },
  {
    "familyName": "Curtis",
    "givenName": "Helen"
  },
  {
    "firstName": "Thomas",
    "relationship": "father"
  },
  {
    "firstName": "Mary Kay",
    "relationship": "mother"
  }
]
```

Following is the query which contains the IN keyword.

```
SELECT *
FROM c IN Families.parents
```

When the above query is executed, it produces the following output.

```
[
  {
    "familyName": "Wakefield",
    "givenName": "Robin"
  },
  {
    "familyName": "Miller",
    "givenName": "Ben"
  },
  {
    "familyName": "Smith",
    "givenName": "James"
  },
  {
    "familyName": "Curtis",
    "givenName": "Helen"
  },
  {
    "firstName": "Thomas",
    "relationship": "father"
  },
  {
    "firstName": "Mary Kay",
    "relationship": "mother"
  }
  {
    "id": "WakefieldFamily",
    "givenName": "Jesse",
    "grade": 6
  }
]
```

In the above example, it can be seen that with iteration, the query that performs iteration over parents in the collection has different output array. Hence, all the parents from each family are added into a single array.

11. DocumentDB SQL – Joins

In relational databases, the Joins clause is used to combine records from two or more tables in a database, and the need to join across tables is very important while designing normalized schemas. Since DocumentDB deals with the denormalized data model of schema-free documents, the JOIN in DocumentDB SQL is the logical equivalent of a "self-join".

Let’s consider the three documents as in the previous examples.

Following is the **AndersenFamily** document.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
            { "firstName": "Thomas", "relationship":  "father" },
            { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
            {
                  "firstName": "Henriette Thaulow",
                  "gender": "female",
                  "grade": 5,
                  "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
            }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

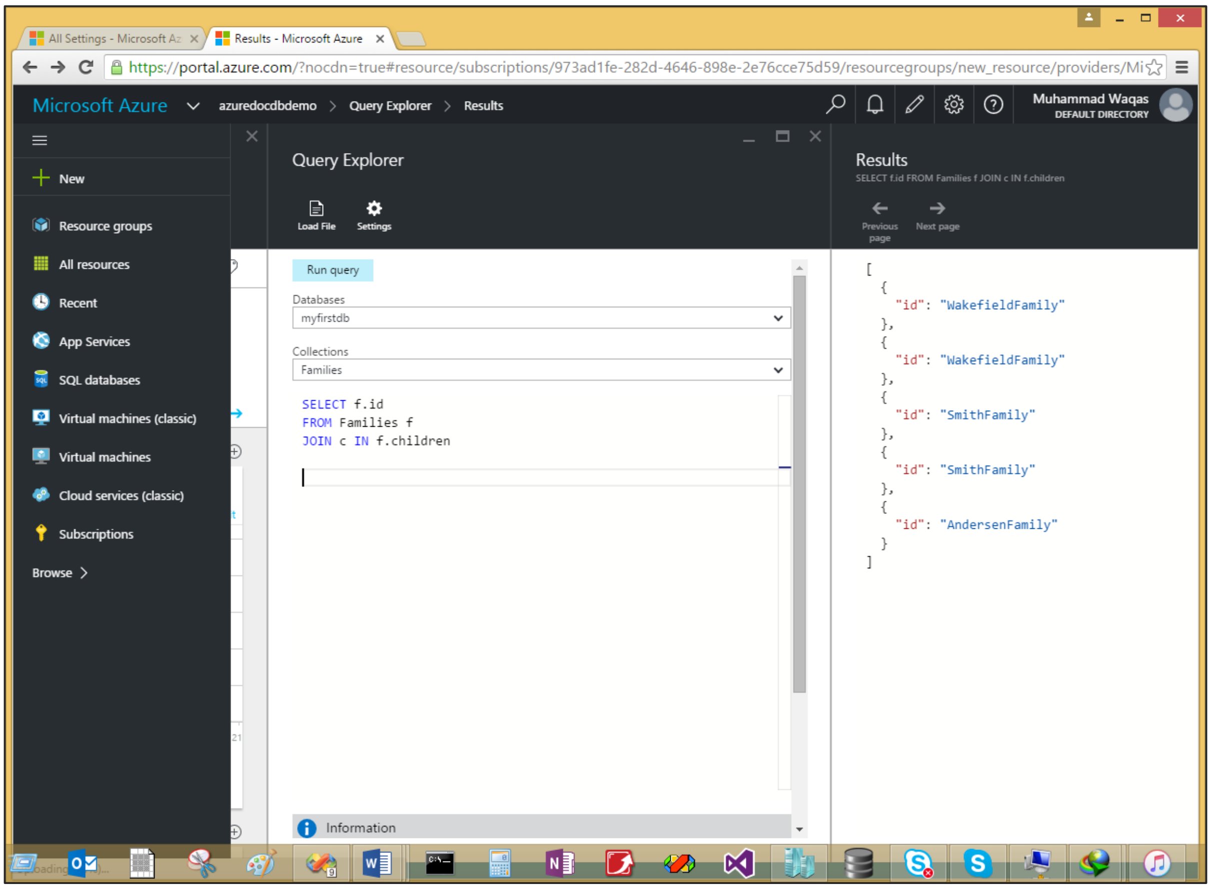
Following is the **SmithFamily** document.

```
{
      "id": "SmithFamily",
      "parents": [
            { "familyName": "Smith", "givenName": "James" },
            { "familyName": "Curtis", "givenName": "Helen" }
      ],
      "children": [
            {
                  "givenName": "Michelle",
                  "gender": "female",
                  "grade": 1
            },
            {
                  "givenName": "John",
                  "gender": "male",
                  "grade": 7,
                  "pets": [
                        { "givenName": "Tweetie", "type": "Bird" }
                  ]
            }
      ],
      "location": {
            "state": "NY",
            "county": "Queens",
            "city": "Forest Hills"
      },
      "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
      "id": "WakefieldFamily",
      "parents": [
            { "familyName": "Wakefield", "givenName": "Robin" },
            { "familyName": "Miller", "givenName": "Ben" }
      ],
      "children": [
            {
                  "familyName": "Merriam",
                  "givenName": "Jesse",
                  "gender": "female",
                  "grade": 6,
                  "pets": [
                        { "givenName": "Charlie Brown", "type": "Dog" },
                { "givenName": "Tiger", "type": "Cat" },
                { "givenName": "Princess", "type": "Cat" }
                  ]
            },
            {
                  "familyName": "Miller",
                  "givenName": "Lisa",
                  "gender": "female",
                  "grade": 3,
                  "pets": [
                        { "givenName": "Jake", "type": "Snake" }
                  ]
            }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```

Let’s take a look at an example to understand how the JOIN clause works.



Databases: myfirstdb

Collections: Families

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

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

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

Following is the query which will join the root to children subdocument.

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

When the above query is executed, it will produce the following output.

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

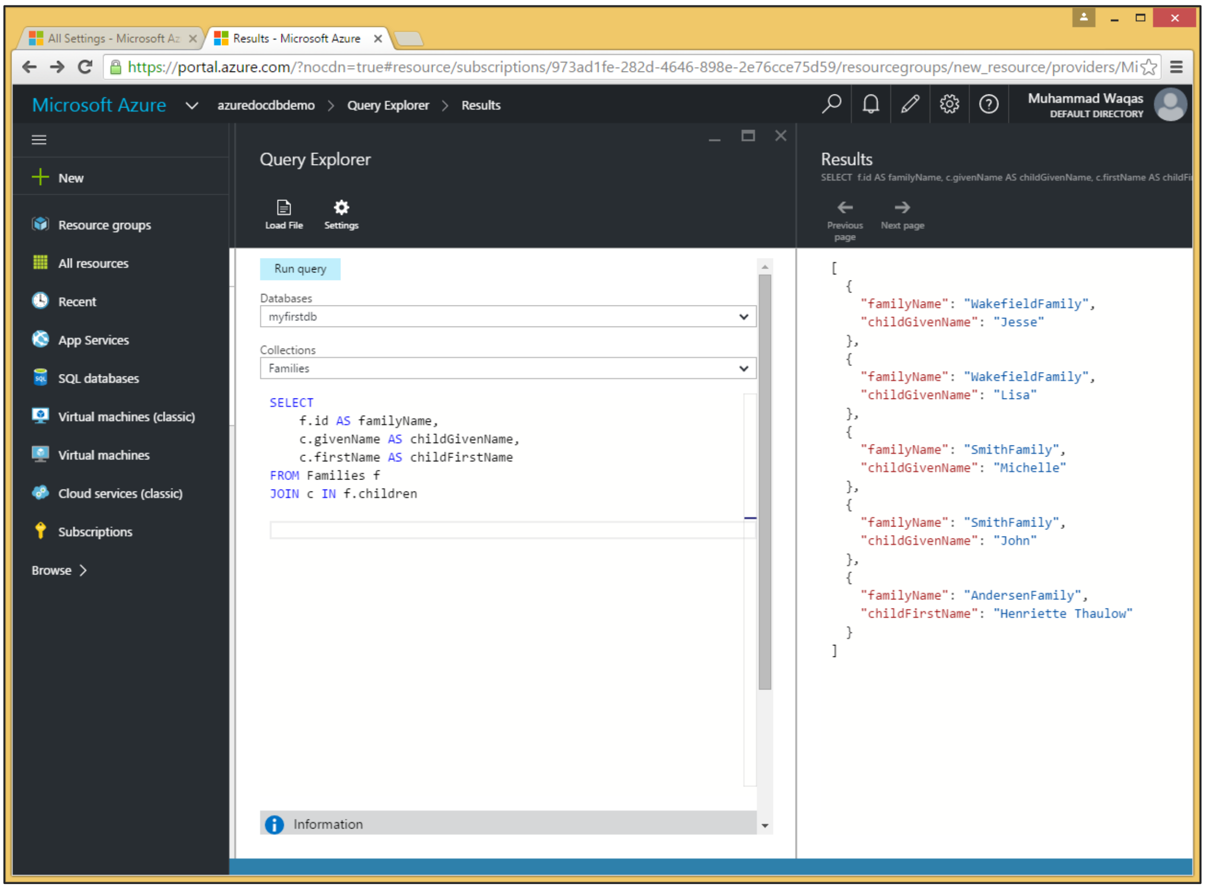
In the above example, the join is between the document root and the children sub-root which makes a cross-product between two JSON objects. Following are certain points to note:

- In the FROM clause, the JOIN clause is an iterator.

- The first two documents WakefieldFamily and SmithFamily contain two children, hence the result set also contains the cross-product which produces a separate object for each child.

- The third document AndersenFamily contains only one children, hence there is only a single object corresponding to this document.

Let's take a look at the same example, however this time we retrieve the child name as well for better understanding of JOIN clause.



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

```
[
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Jesse"
  },
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Lisa"
  },
  {
    "familyName": "SmithFamily",
    "childGivenName": "Michelle"
  },
  {
    "familyName": "SmithFamily",
    "childGivenName": "John"
  },
  {
    "familyName": "AndersenFamily",
    "childFirstName": "Henriette Thaulow"
  }
]
```

Following is the query which will join the root to children subdocument.

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

When the above query is executed, it produces the following output.

```
[
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Jesse"
  },
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Lisa"
  },
  {
    "familyName": "SmithFamily",
    "childGivenName": "Michelle"
  },
  {
    "familyName": "SmithFamily",
    "childGivenName": "John"
  },
  {
    "familyName": "AndersenFamily",
    "childFirstName": "Henriette Thaulow"
  }
]
```

12. DocumentDB SQL – Aliasing

In relational databases, SQL aliases are used to temporarily rename a table or a column heading. Similarly, in DocumentDB, aliases are used to temporarily rename a JSON document, sub-document, object or any field.

The renaming is a temporary change and the actual document does not change. Basically, aliases are created to make field/document names more readable. For aliasing, AS keyword is used which is optional.

Let's consider three similar documents from the ones used in previous examples.

Following is the **AndersenFamily** document.

```
{
     "id": "AndersenFamily",
     "lastName": "Andersen",
     "parents": [
            { "firstName": "Thomas", "relationship":  "father" },
            { "firstName": "Mary Kay", "relationship":  "mother" }
     ],
     "children": [
            {
                   "firstName": "Henriette Thaulow",
                   "gender": "female",
                   "grade": 5,
                   "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
            }
     ],
     "location": { "state": "WA", "county": "King", "city": "Seattle" },
     "isRegistered": true
}
```

Following is the **SmithFamily** document.

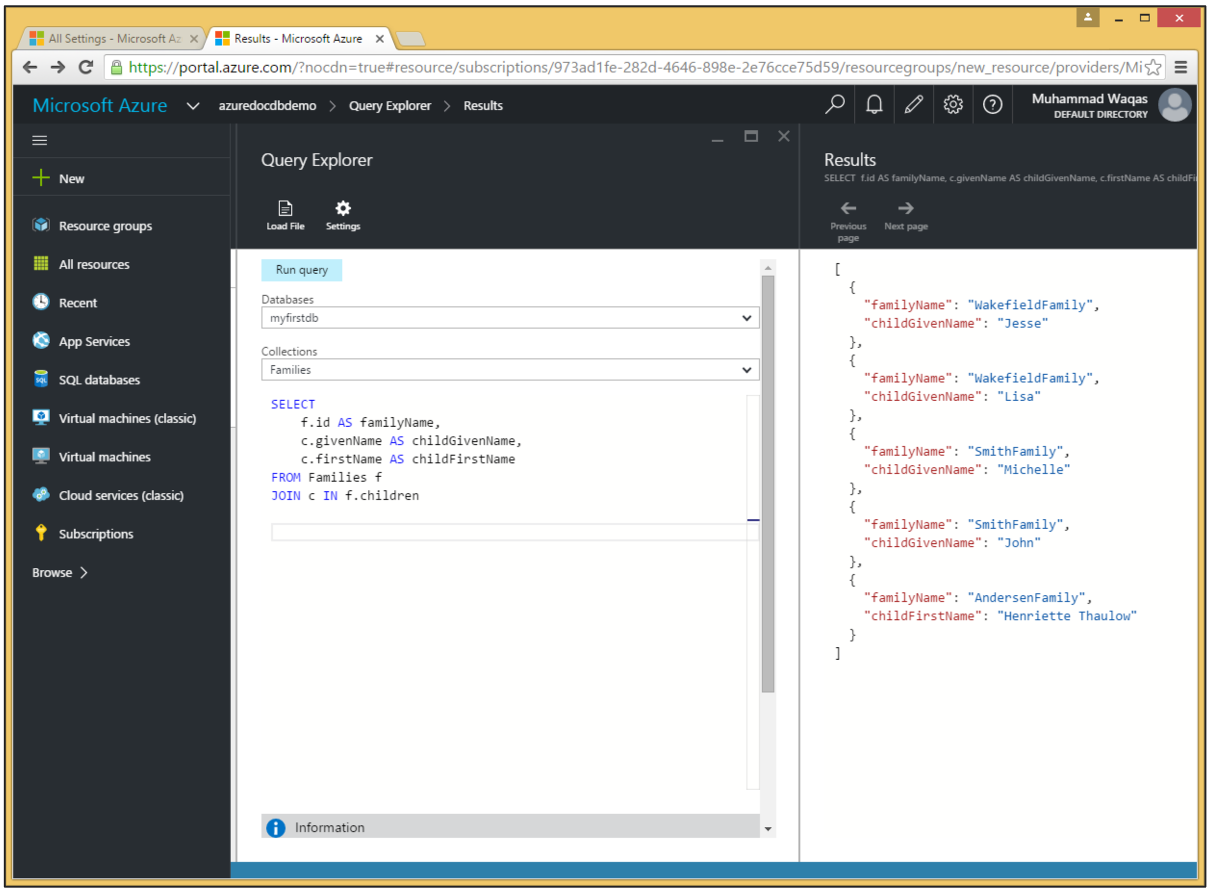
```
{
     "id": "SmithFamily",
     "parents": [
            { "familyName": "Smith", "givenName": "James" },
            { "familyName": "Curtis", "givenName": "Helen" }
```

```
      ],
      "children": [
            {
                  "givenName": "Michelle",
                  "gender": "female",
                  "grade": 1
            },
            {
                  "givenName": "John",
                  "gender": "male",
                  "grade": 7,
                  "pets": [
                        { "givenName": "Tweetie", "type": "Bird" }
                  ]
            }
      ],
      "location": {
            "state": "NY",
            "county": "Queens",
            "city": "Forest Hills"
      },
      "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
      "id": "WakefieldFamily",
      "parents": [
            { "familyName": "Wakefield", "givenName": "Robin" },
            { "familyName": "Miller", "givenName": "Ben" }
      ],
      "children": [
            {
                  "familyName": "Merriam",
                  "givenName": "Jesse",
                  "gender": "female",
                  "grade": 6,
                  "pets": [
                          { "givenName": "Charlie Brown", "type": "Dog" },
               { "givenName": "Tiger", "type": "Cat" },
               { "givenName": "Princess", "type": "Cat" }
                  ]
            },
            {
                  "familyName": "Miller",
                  "givenName": "Lisa",
                  "gender": "female",
                  "grade": 3,
                  "pets": [
                          { "givenName": "Jake", "type": "Snake" }
                  ]
            }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```

Let’s take a look at an example to discuss the aliases.


All Settings - Microsoft Az
Results - Microsoft Azure
https://portal.azure.com/?nocdn=true#resource/subscriptions/973ad1fe-282d-4646-898e-2e76cce75d59/resourcegroups/new_resource/providers/Mi
Microsoft Azure
azuredocdbdemo > Query Explorer > Results
Muhammad Waqas
DEFAULT DIRECTORY
New
Resource groups
All resources
Recent
App Services
SQL databases
Virtual machines (classic)
Virtual machines
Cloud services (classic)
Subscriptions
Browse >
Query Explorer
Load File
Settings
Run query
Information
Results
Previous page
Next page

Databases: myfirstdb

Collections: Families

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

SELECT f.id AS familyName, c.givenName AS childGivenName, c.firstName AS childFi

```
[
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Jesse"
  },
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Lisa"
  },
  {
    "familyName": "SmithFamily",
    "childGivenName": "Michelle"
  },
  {
    "familyName": "SmithFamily",
    "childGivenName": "John"
  },
  {
    "familyName": "AndersenFamily",
    "childFirstName": "Henriette Thaulow"
  }
]
```

Following is the query which will join the root to children subdocument. We have aliases such as f.id AS familyName, c.givenName AS childGivenName, and c.firstName AS childFirstName.

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

When the above query is executed, it produces the following output.

```
[
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Jesse"
  },
  {
    "familyName": "WakefieldFamily",
    "childGivenName": "Lisa"
  },
  {
    "familyName": "SmithFamily",
    "childGivenName": "Michelle"
  },
  {
    "familyName": "SmithFamily",
    "childGivenName": "John"
  },
  {
    "familyName": "AndersenFamily",
    "childFirstName": "Henriette Thaulow"
  }
]
```

The above output shows that the filed names are changed, but it is a temporary change and the original documents are not modified.

13. DocumentDB SQL – Array Creation

In DocumentDB SQL, Microsoft has added a key feature with the help of which we can easily create an array. It means when we run a query, then as a result it will create an array of collection similar to JSON object as a result of query.

Let's consider the same documents as in the previous examples.

Following is the **AndersenFamily** document.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
             { "firstName": "Thomas", "relationship":  "father" },
             { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
             {
                    "firstName": "Henriette Thaulow",
                    "gender": "female",
                    "grade": 5,
                    "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
             }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

Following is the **SmithFamily** document.

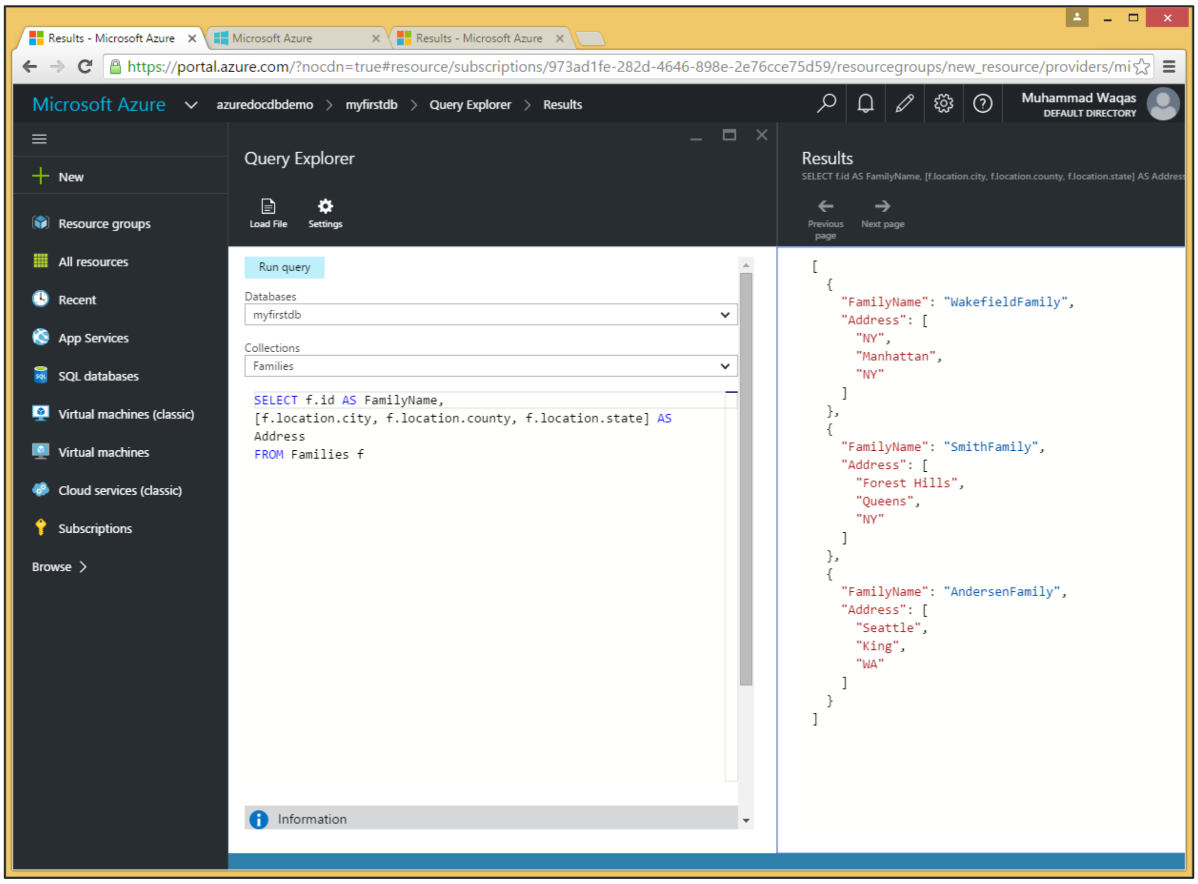
```
{
      "id": "SmithFamily",
      "parents": [
             { "familyName": "Smith", "givenName": "James" },
             { "familyName": "Curtis", "givenName": "Helen" }
      ],
      "children": [
             {
                "givenName": "Michelle",
                "gender": "female",
                "grade": 1
            },
            {
                "givenName": "John",
                "gender": "male",
                "grade": 7,
                "pets": [
                    { "givenName": "Tweetie", "type": "Bird" }
                ]
            }
    ],
    "location": {
        "state": "NY",
        "county": "Queens",
        "city": "Forest Hills"
    },
    "isRegistered": true
}
```

Following is the **WakefieldFamily** document.

```
{
    "id": "WakefieldFamily",
    "parents": [
        { "familyName": "Wakefield", "givenName": "Robin" },
        { "familyName": "Miller", "givenName": "Ben" }
    ],
    "children": [
        {
            "familyName": "Merriam",
            "givenName": "Jesse",
            "gender": "female",
            "grade": 6,
            "pets": [
                { "givenName": "Charlie Brown", "type": "Dog" },
          { "givenName": "Tiger", "type": "Cat" },
          { "givenName": "Princess", "type": "Cat" }
```

```
                    ]
             },
             {
                    "familyName": "Miller",
                    "givenName": "Lisa",
                    "gender": "female",
                    "grade": 3,
                    "pets": [
                           { "givenName": "Jake", "type": "Snake" }
                    ]
             }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```

Let's take a look at an example.



```
SELECT f.id AS FamilyName,
[f.location.city, f.location.county, f.location.state] AS
Address
FROM Families f
```

```
[
  {
    "FamilyName": "WakefieldFamily",
    "Address": [
      "NY",
      "Manhattan",
      "NY"
    ]
  },
  {
    "FamilyName": "SmithFamily",
    "Address": [
      "Forest Hills",
      "Queens",
      "NY"
    ]
  },
  {
    "FamilyName": "AndersenFamily",
    "Address": [
      "Seattle",
      "King",
      "WA"
    ]
  }
]
```

Following is the query which will return the family name and address of each family.

```
SELECT f.id AS FamilyName,
[f.location.city, f.location.county, f.location.state] AS Address
FROM Families f
```

As can be seen city, county and state fields are enclosed in square brackets, which will create an array and this array is named Address. When the above query is executed, it produces the following output.

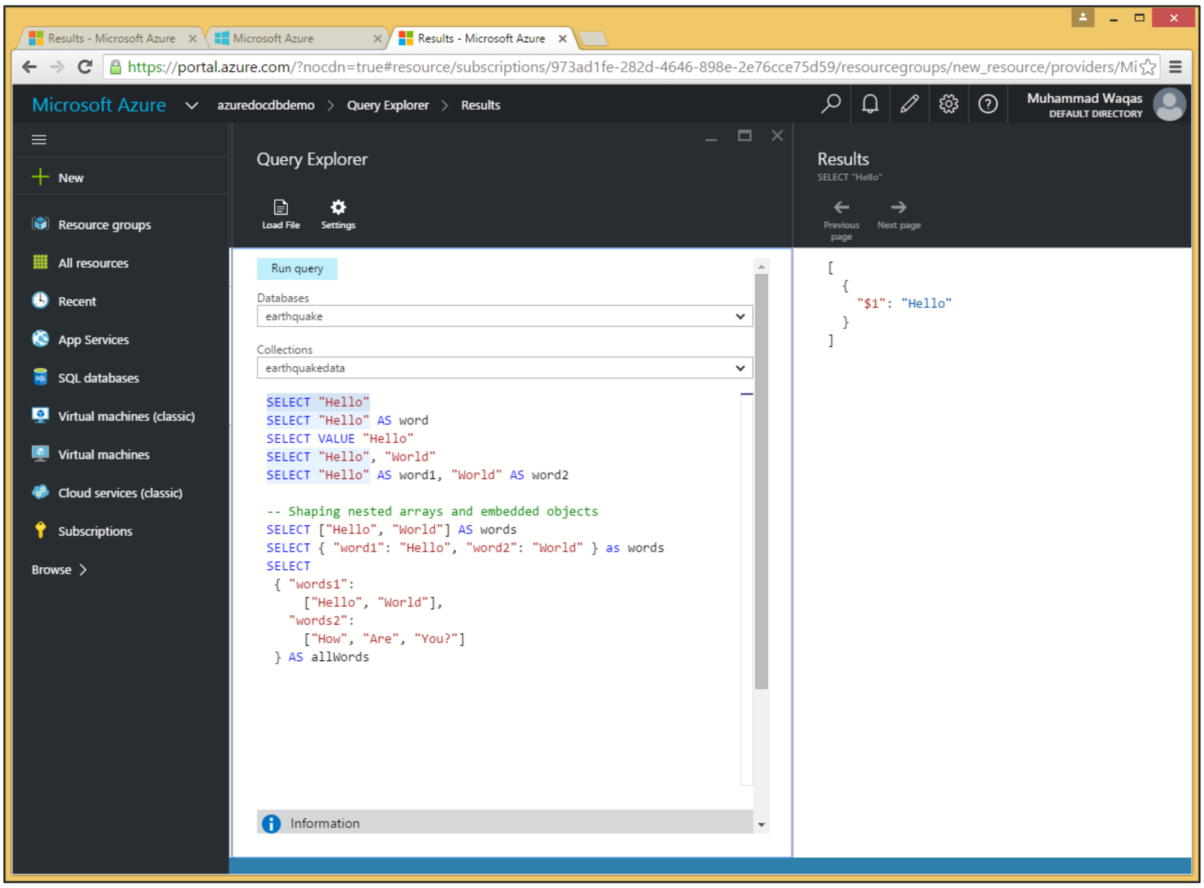
```
[
  {
    "FamilyName": "WakefieldFamily",
    "Address": [
      "NY",
      "Manhattan",
      "NY"
    ]
  },
  {
    "FamilyName": "SmithFamily",
    "Address": [
      "Forest Hills",
      "Queens",
      "NY"
    ]
  },
  {
    "FamilyName": "AndersenFamily",
    "Address": [
      "Seattle",
      "King",
      "WA"
    ]
  }
]
```

The city, county, and state information are added in the Address array in the above output.

14. DocumentDB SQL – Scalar Expressions

In DocumentDB SQL, the SELECT clause also supports scalar expressions like constants, arithmetic expressions, logical expressions, etc. Normally, scalar queries are rarely used, because they don't actually query documents in the collection, they just evaluate expressions. But it's still helpful to use scalar expression queries to learn the basics, how to use expressions and shape JSON in a query, and these concepts apply directly to the actual queries you'll be running against documents in a collection.

Let’s take a look at an example which contains multiple scalar queries.



Databases: earthquake

Collections: earthquakedata

```
SELECT "Hello"
SELECT "Hello" AS word
SELECT VALUE "Hello"
SELECT "Hello", "World"
SELECT "Hello" AS word1, "World" AS word2

-- Shaping nested arrays and embedded objects
SELECT ["Hello", "World"] AS words
SELECT { "word1": "Hello", "word2": "World" } as words
SELECT
 { "words1":
     ["Hello", "World"],
   "words2":
     ["How", "Are", "You?"]
 } AS allWords
```

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

In the Query Explorer, select just the text to be executed and click ‘Run’. Let's run this first one.

```
SELECT "Hello"
```

When the above query is executed, it produces the following output.

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

This output may look a bit confusing, so let's break it down.

- First, as we saw in the last demo, query results are always contained in square brackets because they are returned as a JSON array, even results from scalar expression queries like this one that only returns a single document.

- We have an array with one document in it, and that document has a single property in it for the single expression in the SELECT statement.

- The SELECT statement doesn't provide a name for this property, thus DocumentDB auto generates one using $1.

- This is usually not what we want, which is why we can use AS to alias the expression in the query, which sets the property name in the generated document the way you'd like it to be, word, in this example.

```
SELECT "Hello" AS word
```

When the above query is executed, it produces the following output.

```
[
  {
    "word": "Hello"
  }
]
```

Similarly, following is another simple query.

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

The query retrieves the following output.

```
[
  {
    "$1": 1.3333333333333333
  }
]
```

Let’s take a look at another example of shaping nested arrays and embedded objects.

```
SELECT
 { "words1":
     ["Hello", "World"],
   "words2":
     ["How", "Are", "You?"]
 } AS allWords
```

When the above query is executed, it produces the following output.

```
[
  {
    "allWords": {
      "words1": [
        "Hello",
        "World"
      ],
      "words2": [
        "How",
        "Are",
        "You?"
      ]
    }
  }
]
```

15. DocumentDB SQL – Parameterized SQL

In relational databases, a parameterized query is a query in which placeholders are used for parameters and the parameter values are supplied at execution time. DocumentDB also supports parameterized queries, and parameters in parameterized query can be expressed with the familiar @ notation. The most important reason to use parameterized queries is to avoid SQL injection attacks. It can also provide robust handling and escaping of user input.

Let’s take a look at an example where we will be using the .Net SDK. Following is the code which will delete the collection.

```
private async static Task DeleteCollection(DocumentClient client, string
collectionId)
{
     Console.WriteLine();
     Console.WriteLine(">>> Delete Collection {0} in {1} <<<", collectionId,
_database.Id);


     var query = new SqlQuerySpec
     {
            QueryText = "SELECT * FROM c WHERE c.id = @id",
            Parameters = new SqlParameterCollection { new SqlParameter { Name =
"@id", Value = collectionId } }
     };


     DocumentCollection collection =
client.CreateDocumentCollectionQuery(database.SelfLink,
query).AsEnumerable().First();


     await client.DeleteDocumentCollectionAsync(collection.SelfLink);


     Console.WriteLine("Deleted collection {0} from database {1}",
collectionId, _database.Id);
}
```

The construction of a parameterized query is as follows.

```
var query = new SqlQuerySpec
{
     QueryText = "SELECT * FROM c WHERE c.id = @id",
     Parameters = new SqlParameterCollection { new SqlParameter { Name = "@id", 
Value = collectionId } }
};
```

We are not hardcoding the collectionId so this method can be used to delete any collection. We can use '@' symbol to prefix parameter names, similar to SQL Server.

In the above example, we are querying for a specific collection by Id where the Id parameter is defined in this SqlParameterCollection assigned to the parameter's property of this SqlQuerySpec. The SDK then does the work of constructing the final query string for DocumentDB with the collectionId embedded inside of it. We run the query and then use its SelfLink to delete the collection.

Following is the CreateDocumentClient task implementation.

```
private static async Task CreateDocumentClient()
{
    // Create a new instance of the DocumentClient
    using (var client = new DocumentClient(new Uri(EndpointUrl), 
AuthorizationKey))
    {
        database = client.CreateDatabaseQuery("SELECT * FROM c WHERE c.id = 
'earthquake'").AsEnumerable().First();
        collection = 
client.CreateDocumentCollectionQuery(database.CollectionsLink, "SELECT * FROM c 
WHERE c.id = 'myfirstdb'").AsEnumerable().First();


        await DeleteCollection(client, "MyCollection1");
        await DeleteCollection(client, "MyCollection2");
    }
}
```

When the code is executed, it produces the following output.

```
**** Delete Collection MyCollection1 in mydb ****
Deleted collection MyCollection1 from database myfirstdb


**** Delete Collection MyCollection2 in mydb ****
Deleted collection MyCollection2 from database myfirstdb
```

Let's take a look at another example. We can write a query that takes last name and address state as parameters, and then executes it for various values of lastname and location.state based on the user input.

```
SELECT *
FROM Families f
WHERE f.lastName = @lastName AND f.location.state = @addressState
```

This request can then be sent to DocumentDB as a parameterized JSON query as shown in the following code.

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

16. DocumentDB SQL – Built-in Functions

DocumentDB supports a host of built-in functions for common operations that can be used inside queries. There are a bunch of functions for performing mathematical calculations, and also type checking functions that are extremely useful while working with varying schemas. These functions can test if a certain property exists and if it does whether it's a number or a string, Boolean or object.

We also get these handy functions for parsing and manipulating strings, as well as several functions for working with arrays allowing you to do things like concatenate arrays and test to see if an array contains a particular element.

Following are the different types of built-in functions:

- Mathematical functions
- Type checking functions
- String functions
- Array functions
- Spatial functions

Mathematical Functions

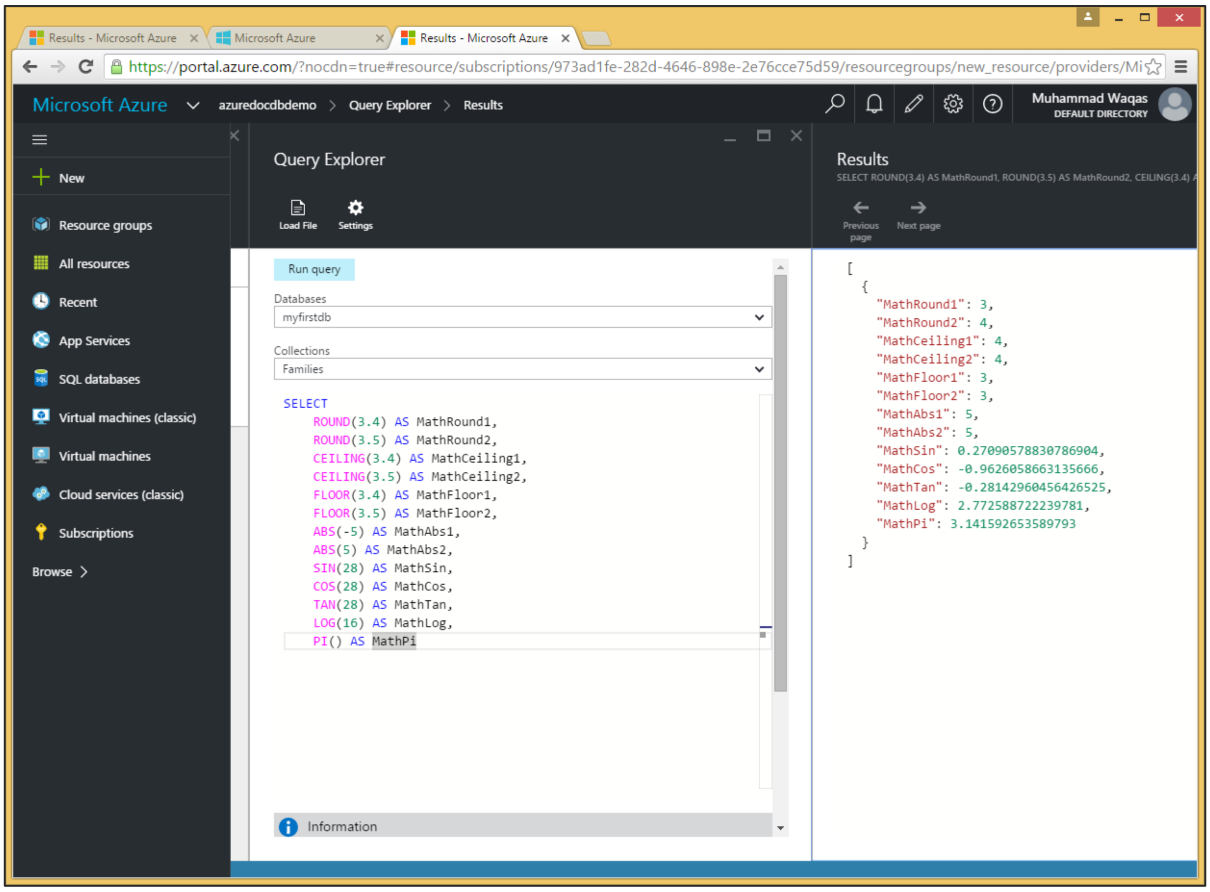
The mathematical functions perform a calculation, usually based on input values that are provided as arguments, and return a numeric value. Following are the supported built-in mathematical functions in DocumentDB.

Function	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 specified.
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.

Let's take a look at an example where we will be using some built-in mathematical functions.



Databases: myfirstdb

Collections: Families

```
SELECT
    ROUND(3.4) AS MathRound1,
    ROUND(3.5) AS MathRound2,
    CEILING(3.4) AS MathCeiling1,
    CEILING(3.5) AS MathCeiling2,
    FLOOR(3.4) AS MathFloor1,
    FLOOR(3.5) AS MathFloor2,
    ABS(-5) AS MathAbs1,
    ABS(5) AS MathAbs2,
    SIN(28) AS MathSin,
    COS(28) AS MathCos,
    TAN(28) AS MathTan,
    LOG(16) AS MathLog,
    PI() AS MathPi
```

SELECT ROUND(3.4) AS MathRound1, ROUND(3.5) AS MathRound2, CEILING(3.4) A

```
[
  {
    "MathRound1": 3,
    "MathRound2": 4,
    "MathCeiling1": 4,
    "MathCeiling2": 4,
    "MathFloor1": 3,
    "MathFloor2": 3,
    "MathAbs1": 5,
    "MathAbs2": 5,
    "MathSin": 0.27090578830786904,
    "MathCos": -0.9626058663135666,
    "MathTan": -0.28142960456426525,
    "MathLog": 2.772588722239781,
    "MathPi": 3.141592653589793
  }
]
```

Following is a query in which you can see the numeric rounding functions, ROUND, CEILING, FLOOR, as well as the absolute value, sin, cosign, tangent, natural logarithm, and pi functions.

```
SELECT

     ROUND(3.4) AS MathRound1,

     ROUND(3.5) AS MathRound2,

     CEILING(3.4) AS MathCeiling1,

     CEILING(3.5) AS MathCeiling2,

     FLOOR(3.4) AS MathFloor1,

     FLOOR(3.5) AS MathFloor2,

     ABS(-5) AS MathAbs1,

     ABS(5) AS MathAbs2,

     SIN(28) AS MathSin,

     COS(28) AS MathCos,

     TAN(28) AS MathTan,

     LOG(16) AS MathLog,

     PI() AS MathPi
```

When the above query is executed, it produces the following output.

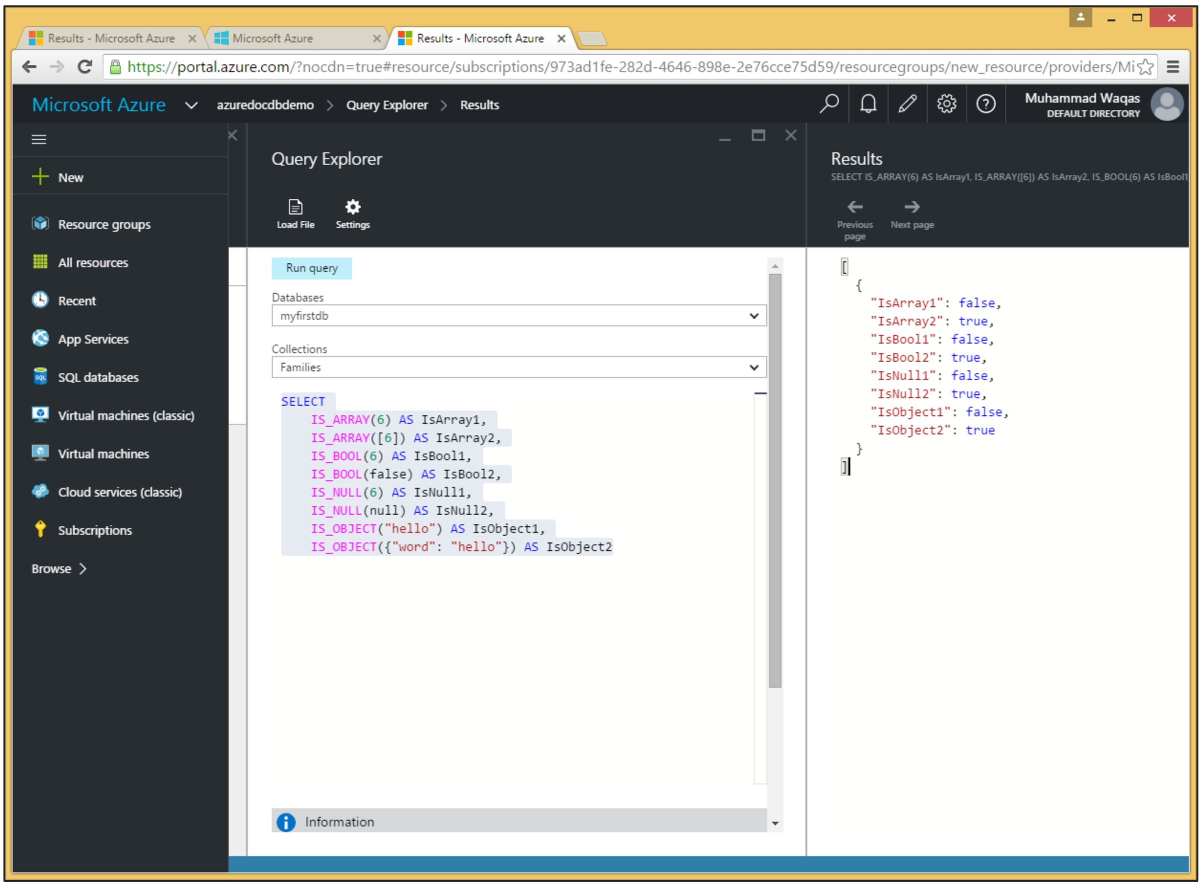
```
[

  {

    "MathRound1": 3,

    "MathRound2": 4,

    "MathCeiling1": 4,

    "MathCeiling2": 4,

    "MathFloor1": 3,

    "MathFloor2": 3,

    "MathAbs1": 5,

    "MathAbs2": 5,

    "MathSin": 0.27090578830786904,

    "MathCos": -0.9626058663135666,
    "MathTan": -0.28142960456426525,

    "MathLog": 2.772588722239781,

    "MathPi": 3.141592653589793

  }

]
```

Type Checking Functions

The type checking functions allow you to check the type of an expression within SQL queries. It can be used to determine the type of properties within documents on the fly when it is variable or unknown. Following are the supported built-in type checking functions.

Function	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.

Let's take a look at another example where some built-in type checking functions are used.



```
SELECT
    IS_ARRAY(6) AS IsArray1,
    IS_ARRAY([6]) AS IsArray2,
    IS_BOOL(6) AS IsBool1,
    IS_BOOL(false) AS IsBool2,
    IS_NULL(6) AS IsNull1,
    IS_NULL(null) AS IsNull2,
    IS_OBJECT("hello") AS IsObject1,
    IS_OBJECT({"word": "hello"}) AS IsObject2
```

```
[
  {
    "IsArray1": false,
    "IsArray2": true,
    "IsBool1": false,
    "IsBool2": true,
    "IsNull1": false,
    "IsNull2": true,
    "IsObject1": false,
    "IsObject2": true
  }
]
```

Following is the query with type checking functions.

```
SELECT
     IS_ARRAY(6) AS IsArray1,
     IS_ARRAY([6]) AS IsArray2,
     IS_BOOL(6) AS IsBool1,
     IS_BOOL(false) AS IsBool2,
     IS_NULL(6) AS IsNull1,
     IS_NULL(null) AS IsNull2,
     IS_OBJECT("hello") AS IsObject1,
     IS_OBJECT({"word": "hello"}) AS IsObject2
```

When the above query is executed, it produces the following output.

```
[
  {
    "IsArray1": false,
    "IsArray2": true,
    "IsBool1": false,
    "IsBool2": true,
    "IsNull1": false,
    "IsNull2": true,
    "IsObject1": false,
    "IsObject2": true
  }
]
```

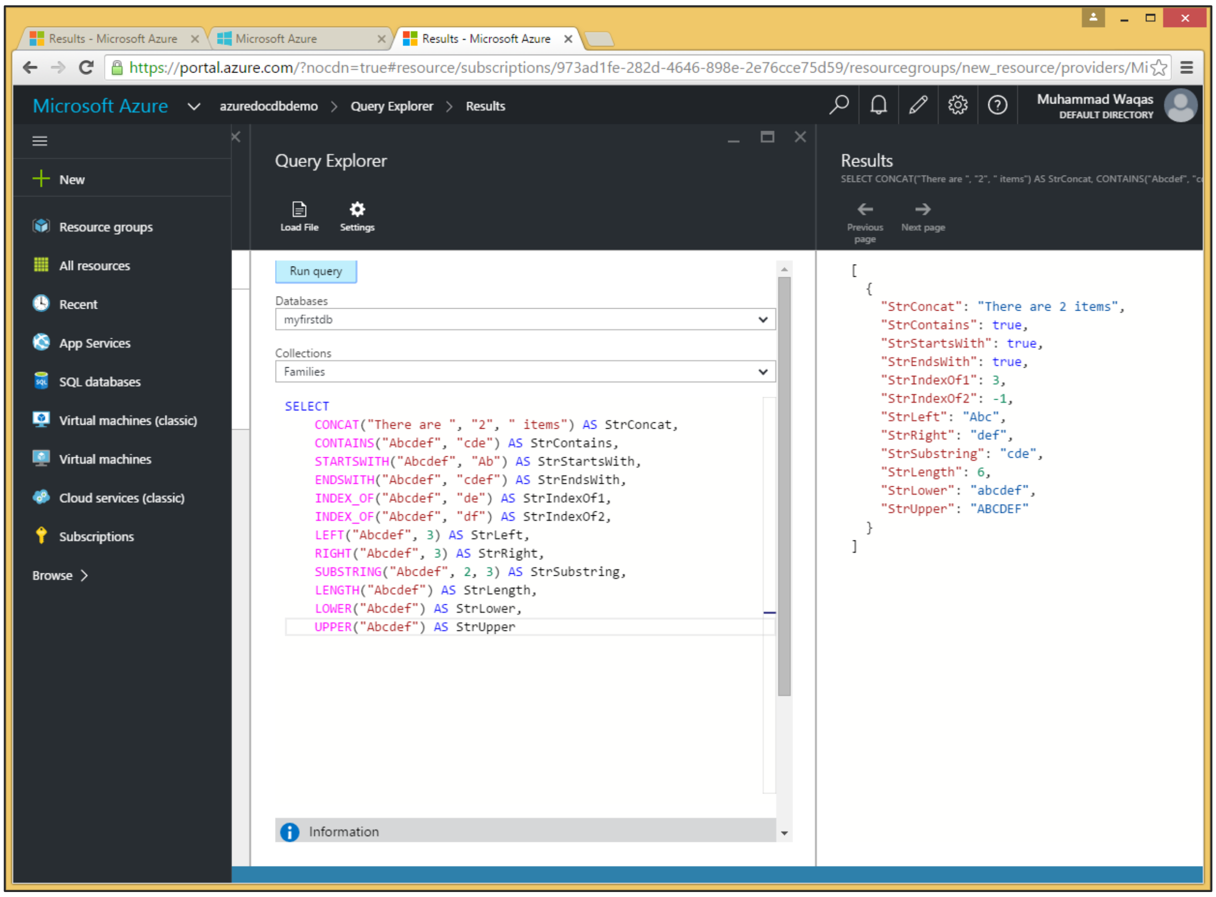
String Functions

The string functions perform an operation on a string input value and return a string, numeric or Boolean value. Following are the supported built-in string functions.

Function	Description
LENGTH (str_expr)	Returns the number of characters of the specified string expression.
CONCAT (str_expr, str_expr [, str_expr])	Returns a string that is the result of concatenating two or more string values.
SUBSTRING (str_expr, num_expr, num_expr)	Returns part of a string expression.
STARTSWITH (str_expr, str_expr)	Returns a Boolean indicating whether the first string expression ends with the second.
ENDSWITH (str_expr, str_expr)	Returns a Boolean indicating whether the first string expression ends with the second.
CONTAINS (str_expr, str_expr)	Returns a Boolean indicating whether the first string expression contains the second.
INDEX_OF (str_expr, str_expr)	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)	Returns the left part of a string with the specified number of characters.
RIGHT (str_expr, num_expr)	Returns the right part of a string with the specified number of characters.

LTRIM (str_expr)	Returns a string expression after it removes leading blanks.
RTRIM (str_expr)	Returns a string expression after truncating all trailing blanks.
LOWER (str_expr)	Returns a string expression after converting uppercase character data to lowercase.
UPPER (str_expr)	Returns a string expression after converting lowercase character data to uppercase.
REPLACE (str_expr, str_expr, str_expr)	Replaces all occurrences of a specified string value with another string value.
REPLICATE (str_expr, num_expr)	Repeats a string value a specified number of times.
REVERSE (str_expr)	Returns the reverse order of a string value.

Let's take a look at an example where some built-in string functions are used.



```
SELECT
    CONCAT("There are ", "2", " items") AS StrConcat,
    CONTAINS("Abcdef", "cde") AS StrContains,
    STARTSWITH("Abcdef", "Ab") AS StrStartsWith,
    ENDSWITH("Abcdef", "cdef") AS StrEndsWith,
    INDEX_OF("Abcdef", "de") AS StrIndexOf1,
    INDEX_OF("Abcdef", "df") AS StrIndexOf2,
    LEFT("Abcdef", 3) AS StrLeft,
    RIGHT("Abcdef", 3) AS StrRight,
    SUBSTRING("Abcdef", 2, 3) AS StrSubstring,
    LENGTH("Abcdef") AS StrLength,
    LOWER("Abcdef") AS StrLower,
    UPPER("Abcdef") AS StrUpper
```

```
[
  {
    "StrConcat": "There are 2 items",
    "StrContains": true,
    "StrStartsWith": true,
    "StrEndsWith": true,
    "StrIndexOf1": 3,
    "StrIndexOf2": -1,
    "StrLeft": "Abc",
    "StrRight": "def",
    "StrSubstring": "cde",
    "StrLength": 6,
    "StrLower": "abcdef",
    "StrUpper": "ABCDEF"
  }
]
```

Following is the query with different string functions.

```
SELECT
      CONCAT("There are ", "2", " items") AS StrConcat,
      CONTAINS("Abcdef", "cde") AS StrContains,
      STARTSWITH("Abcdef", "Ab") AS StrStartsWith,
      ENDSWITH("Abcdef", "cdef") AS StrEndsWith,
      INDEX_OF("Abcdef", "de") AS StrIndexOf1,
      INDEX_OF("Abcdef", "df") AS StrIndexOf2,
      LEFT("Abcdef", 3) AS StrLeft,
      RIGHT("Abcdef", 3) AS StrRight,
      SUBSTRING("Abcdef", 2, 3) AS StrSubstring,
      LENGTH("Abcdef") AS StrLength,
      LOWER("Abcdef") AS StrLower,
      UPPER("Abcdef") AS StrUpper
```

When the above query is executed, it produces the following output.

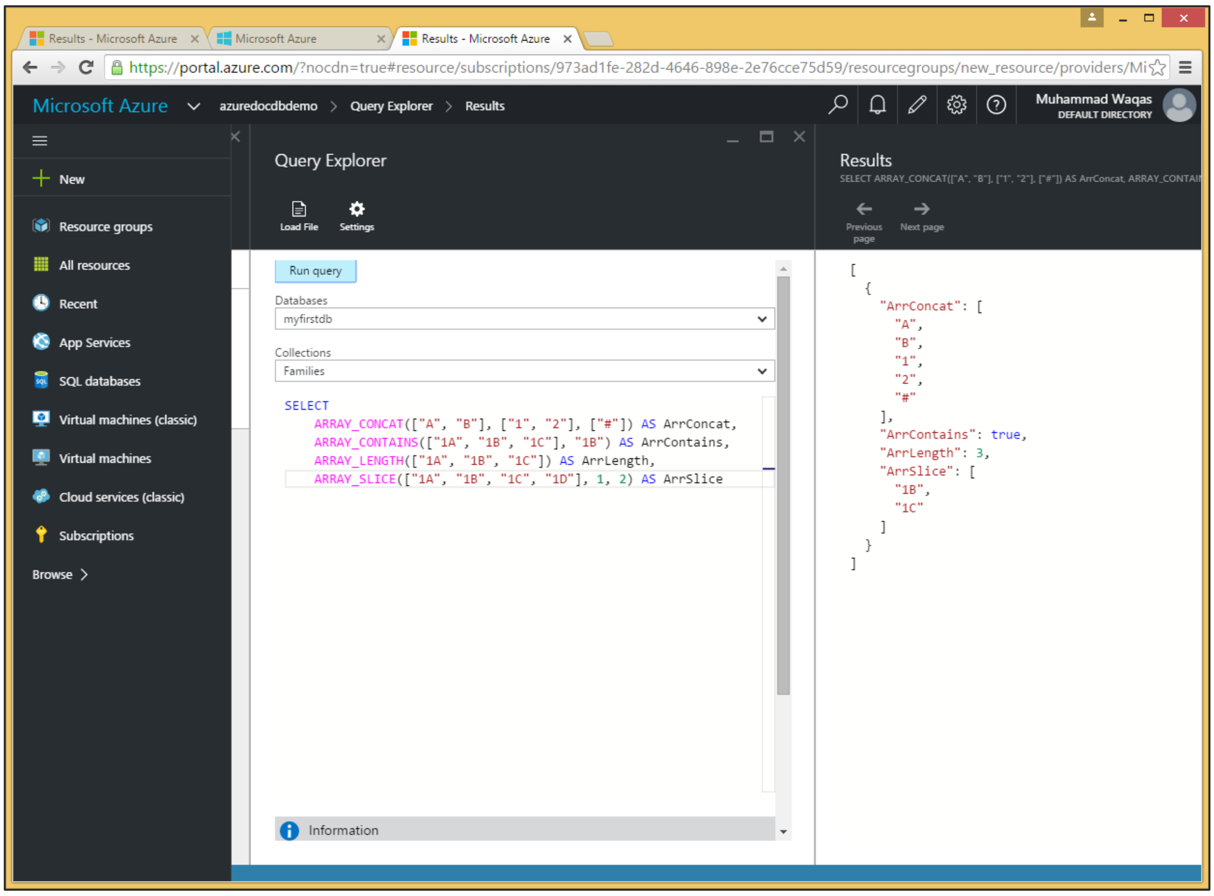
```
[
  {
    "StrConcat": "There are 2 items",
    "StrContains": true,
    "StrStartsWith": true,
    "StrEndsWith": true,
    "StrIndexOf1": 3,
    "StrIndexOf2": -1,
    "StrLeft": "Abc",
    "StrRight": "def",
    "StrSubstring": "cde",
    "StrLength": 6,
    "StrLower": "abcdef",
    "StrUpper": "ABCDEF"
  }
]
```

Array Functions

The array functions perform an operation on an array input value and return in the form of numeric, Boolean or array value. Following are the built-in array functions.

Function	Description
ARRAY_LENGTH (arr_expr)	Returns the number of elements of the specified array expression.
ARRAY_CONCAT (arr_expr, arr_expr [, arr_expr])	Returns an array that is the result of concatenating two or more array values.
ARRAY_CONTAINS (arr_expr, expr)	Returns a Boolean indicating whether the array contains the specified value.
ARRAY_SLICE (arr_expr, num_expr [, num_expr])	Returns part of an array expression.

Let's take a look at another example where some built-in array functions are used.



Results - Microsoft Azure | Microsoft Azure | Results - Microsoft Azure

https://portal.azure.com/?nocdn=true#resource/subscriptions/973ad1fe-282d-4646-898e-2e76cce75d59/resourcegroups/new_resource/providers/Mi

Microsoft Azure > azuredocdbdemo > Query Explorer > Results

Muhammad Waqas
DEFAULT DIRECTORY

Query Explorer

Databases: myfirstdb

Collections: Families

```
SELECT
    ARRAY_CONCAT(["A", "B"], ["1", "2"], ["#"]) AS ArrConcat,
    ARRAY_CONTAINS(["1A", "1B", "1C"], "1B") AS ArrContains,
    ARRAY_LENGTH(["1A", "1B", "1C"]) AS ArrLength,
    ARRAY_SLICE(["1A", "1B", "1C", "1D"], 1, 2) AS ArrSlice
```

Results

SELECT ARRAY_CONCAT(["A", "B"], ["1", "2"], ["#"]) AS ArrConcat, ARRAY_CONTAI

```
[
  {
    "ArrConcat": [
      "A",
      "B",
      "1",
      "2",
      "#"
    ],
    "ArrContains": true,
    "ArrLength": 3,
    "ArrSlice": [
      "1B",
      "1C"
    ]
  }
]
```

Following is the query with different array functions.

```
SELECT
     ARRAY_CONCAT(["A", "B"], ["1", "2"], ["#"]) AS ArrConcat,
     ARRAY_CONTAINS(["1A", "1B", "1C"], "1B") AS ArrContains,
     ARRAY_LENGTH(["1A", "1B", "1C"]) AS ArrLength,
     ARRAY_SLICE(["1A", "1B", "1C", "1D"], 1, 2) AS ArrSlice
```

When the above query is executed, it produces the following output.

```
[
  {
    "ArrConcat": [
      "A",
      "B",
      "1",
      "2",
      "#"
    ],
    "ArrContains": true,
    "ArrLength": 3,
    "ArrSlice": [
      "1B",
      "1C"
    ]
  }
]
```

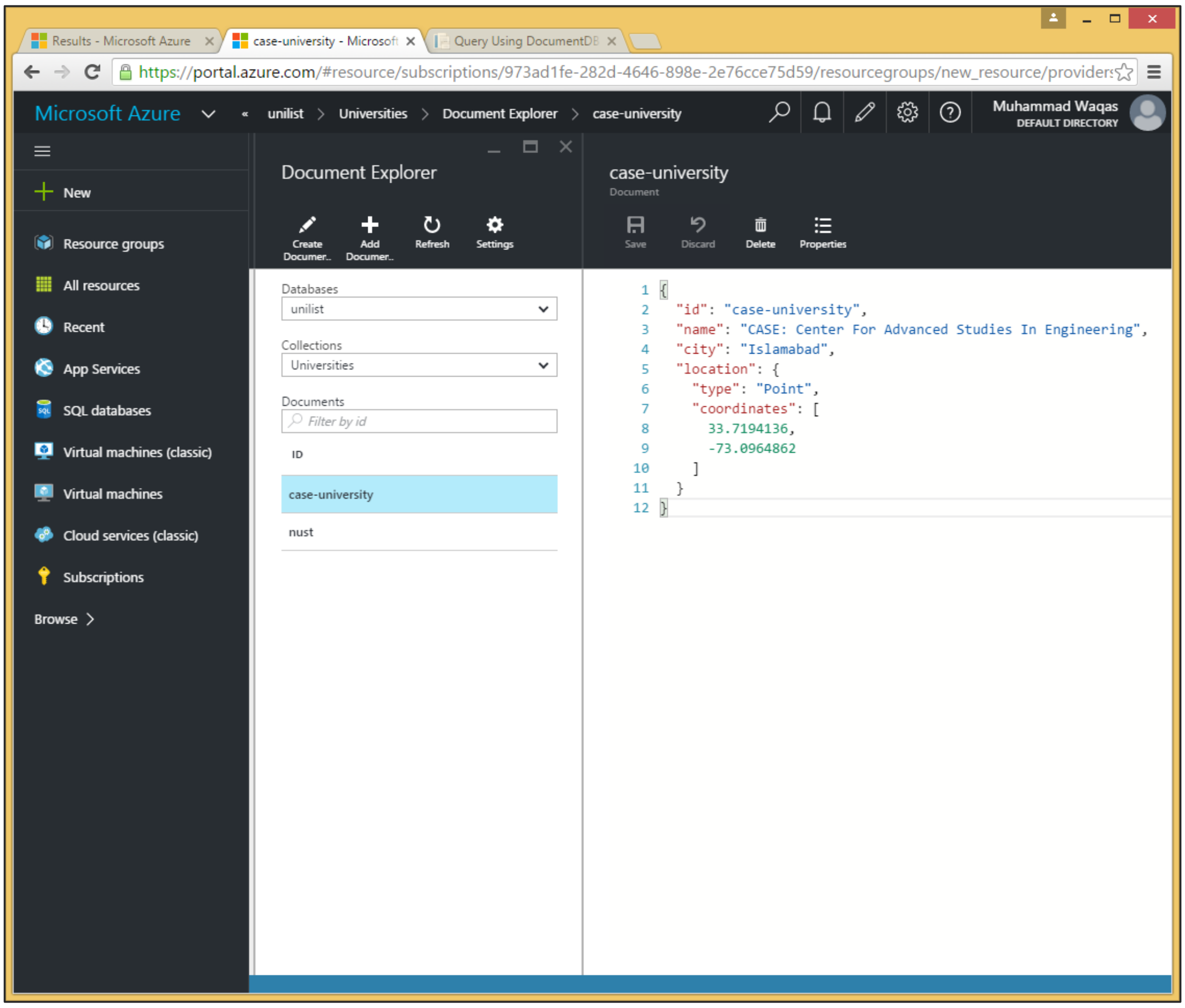
Spatial Functions

DocumentDB also supports the Open Geospatial Consortium (OGC) built-in functions for geospatial querying. Following is a list of built-in supported spatial functions.

Function	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.

In this example, we will use the following two documents of universities which contains the location in the form of coordinates.

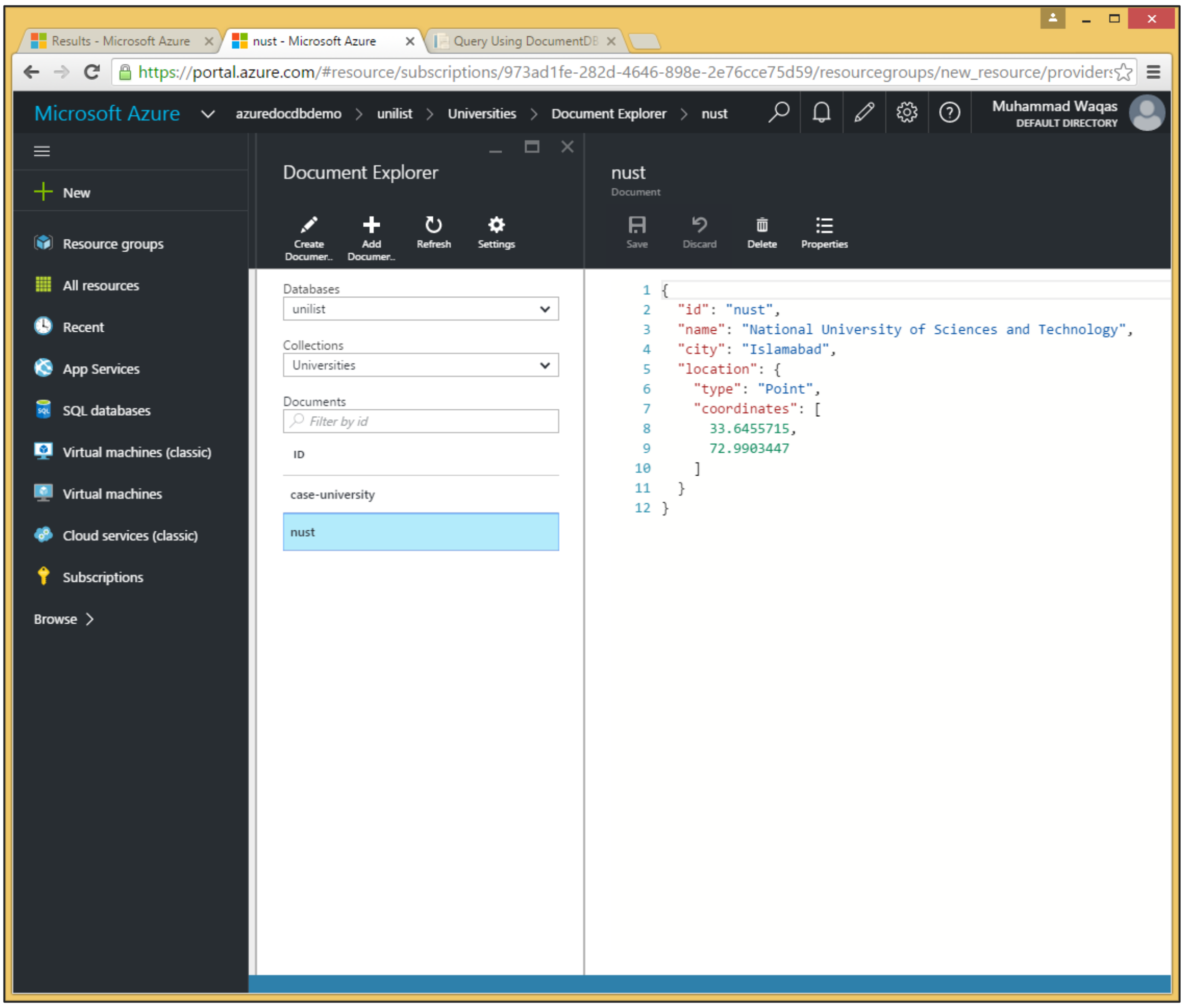
Following is the **Case University document**.



```
{
  "id": "case-university",
  "name": "CASE: Center For Advanced Studies In Engineering",
  "city": "Islamabad",
  "location": {
    "type": "Point",
    "coordinates": [
      33.7194136,
      -73.0964862
    ]
  }
}
```

```
{  "id": "case-university",
  "name": "CASE: Center For Advanced Studies In Engineering",
  "city": "Islamabad",
  "location": {
    "type": "Point",
    "coordinates": [
      33.7194136,
      -73.0964862
    ]
  }
}
```

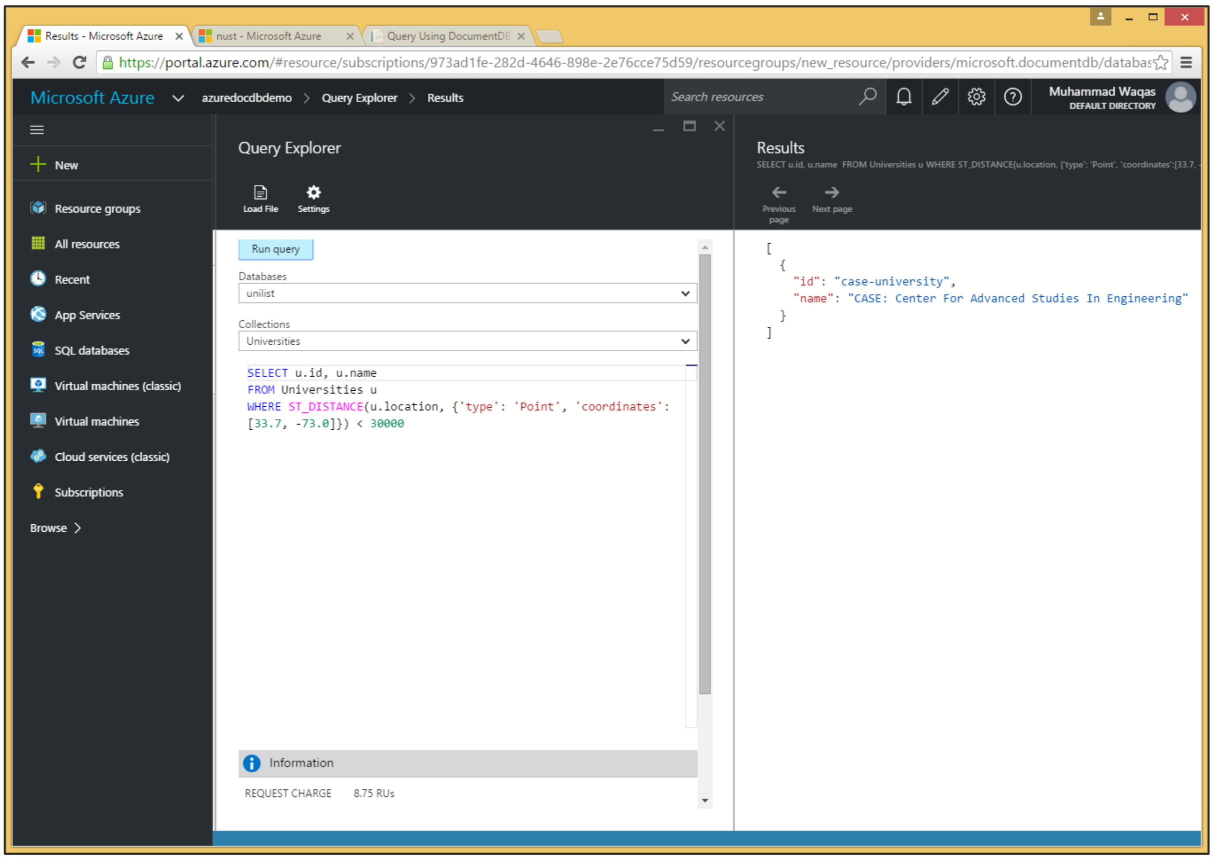
Following is the **Nust University document**.



```
{
  "id": "nust",
  "name": "National University of Sciences and Technology",
  "city": "Islamabad",
  "location": {
    "type": "Point",
    "coordinates": [
      33.6455715,
      72.9903447
    ]
  }
}
```

```
{
  "id": "nust",
  "name": "National University of Sciences and Technology",
  "city": "Islamabad",
  "location": {
    "type": "Point",
    "coordinates": [
      33.6455715,
      72.9903447
    ]
  }
}
```

Let’s take a look at another example of ST_DISTANCE.



Query Explorer

Databases: unilist

Collections: Universities

```
SELECT u.id, u.name
FROM Universities u
WHERE ST_DISTANCE(u.location, {'type': 'Point', 'coordinates':
[33.7, -73.0]}) < 30000
```

REQUEST CHARGE 8.75 RUs

Results

SELECT u.id, u.name FROM Universities u WHERE ST_DISTANCE(u.location, {'type': 'Point', 'coordinates':[33.7,

```
[
  {
    "id": "case-university",
    "name": "CASE: Center For Advanced Studies In Engineering"
  }
]
```

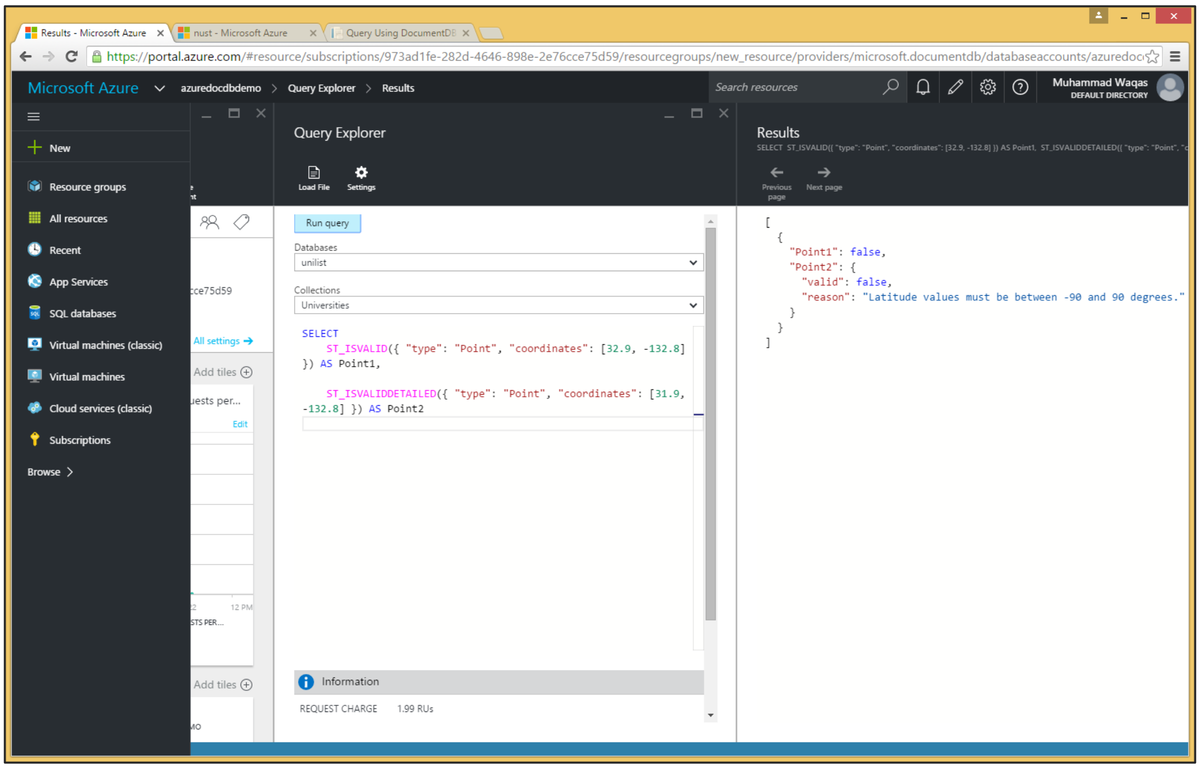
Following is the query that returns id and name of the universities documents that are within 30 km of the specified location.

```
SELECT u.id, u.name

FROM Universities u

WHERE ST_DISTANCE(u.location, {'type': 'Point', 'coordinates':[33.7, -73.0]}) <
30000
```

When the above query is executed, it produces the following output.

```
[

  {

    "id": "case-university",

    "name": "CASE: Center For Advanced Studies In Engineering"

  }

]
```

Let’s take a look at another example.



Databases: unilist

Collections: Universities

```
SELECT
    ST_ISVALID({ "type": "Point", "coordinates": [32.9, -132.8]
}) AS Point1,

    ST_ISVALIDDETAILED({ "type": "Point", "coordinates": [31.9,
-132.8] }) AS Point2
```

REQUEST CHARGE 1.99 RUs

```
[
  {
    "Point1": false,
    "Point2": {
      "valid": false,
      "reason": "Latitude values must be between -90 and 90 degrees."
    }
  }
]
```

Following is the query which contains ST_ISVALID and ST_ISVALIDDETAILED.

```
SELECT
     ST_ISVALID({ "type": "Point", "coordinates": [32.9, -132.8] }) AS Point1,

     ST_ISVALIDDETAILED({ "type": "Point", "coordinates": [31.9, -132.8] }) AS
Point2
```

When the above query is executed, it produces the following output.

```
[
  {    "Point1": false,
    "Point2": {
      "valid": false,
      "reason": "Latitude values must be between -90 and 90 degrees."
    }
  }]
```

The above output shows that ST_ISVALIDDETAILED also returns the reason why this point is invalid, but ST_ISVALID only returns the Boolean value.

17. DocumentDB SQL – LINQ to SQL Translation

In DocumentDB, we actually use SQL to query documents. If we are doing .NET development, there is also a LINQ provider that can be used and which can generate appropriate SQL from a LINQ query.

Supported Data Types

In DocumentDB, all JSON primitive types are supported in the LINQ provider included with the DocumentDB .NET SDK which are as follows:

- Numeric
- Boolean
- String
- Null

Supported Expression

The following scalar expressions are supported in the LINQ provider included with the DocumentDB .NET SDK.

- **Constant Values**: Includes constant values of the primitive data types.
- **Property/Array Index Expressions:** Expressions refer to the property of an object or an array element.
- **Arithmetic Expressions**: Includes common arithmetic expressions on numerical and Boolean values.
- **String Comparison Expression:** Includes comparing a string value to some constant string value.
- **Object/Array Creation Expression:** Returns an object of compound value type or anonymous type or an array of such objects. These values can be nested.

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 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 Extension Function**: Supports translation from the stub method UserDefinedFunctionProvider.Invoke to the corresponding user-defined function.
- **Miscellaneous**: Supports translation of coalesce and conditional operators. Can translate Contains to String CONTAINS, ARRAY_CONTAINS or the SQL IN depending on context.

Let's take a look at an example where we will be using the .Net SDK. Following are the three documents which we will be consider for this example.

New Customer 1:

```
{
  "name": "New Customer 1",
  "address": {
    "addressType": "Main Office",
    "addressLine1": "123 Main Street",
    "location": {
      "city": "Brooklyn",
      "stateProvinceName": "New York"
    },
    "postalCode": "11229",
    "countryRegionName": "United States"
  },
}
```

New Customer 2:

```
{
  "name": "New Customer 2",
  "address": {
    "addressType": "Main Office",
    "addressLine1": "678 Main Street",
    "location": {
      "city": "London",
      "stateProvinceName": " London "
    },
    "postalCode": "11229",
    "countryRegionName": "United Kingdom"
  },
}
```

New Customer 3:

```
{
  "name": "New Customer 3",
  "address": {
    "addressType": "Main Office",
    "addressLine1": "12 Main Street",
    "location": {
      "city": "Brooklyn",
      "stateProvinceName": "New York"
    },
    "postalCode": "11229",
    "countryRegionName": "United States"
  },
```

Following is the code in which we query using LINQ. We've defined a LINQ query in **q**, but it won't execute until we run .ToList on it.

```
private static void QueryDocumentsWithLinq(DocumentClient client)
{
    Console.WriteLine();
    Console.WriteLine("**** Query Documents (LINQ) ****");
    Console.WriteLine();

    Console.WriteLine("Quering for US customers (LINQ)");
    var q =
        from d in
client.CreateDocumentQuery<Customer>(collection.DocumentsLink)
        where d.Address.CountryRegionName == "United States"
        select new
        {
            Id = d.Id,
            Name = d.Name,
            City = d.Address.Location.City
        };

    var documents = q.ToList();

    Console.WriteLine("Found {0} US customers", documents.Count);
    foreach (var document in documents)
    {
        var d = document as dynamic;
        Console.WriteLine(" Id: {0}; Name: {1}; City: {2}", d.Id, d.Name, d.City);
    }
    Console.WriteLine();
}
```

The SDK will convert our LINQ query into SQL syntax for DocumentDB, generating a SELECT and WHERE clause based on our LINQ syntax.

Let’s call the above queries from the CreateDocumentClient task.

```
private static async Task CreateDocumentClient()
{
    // Create a new instance of the DocumentClient
    using (var client = new DocumentClient(new Uri(EndpointUrl),
AuthorizationKey))
    {
        database = client.CreateDatabaseQuery("SELECT * FROM c WHERE c.id =
'myfirstdb'").AsEnumerable().First();
        collection =
client.CreateDocumentCollectionQuery(database.CollectionsLink, "SELECT * FROM c
WHERE c.id = 'MyCollection'").AsEnumerable().First();


        QueryDocumentsWithLinq(client);
    }
}
```

When the above code is executed, it produces the following output.

```
**** Query Documents (LINQ) ****


Quering for US customers (LINQ)
Found 2 US customers
 Id: 7e9ad4fa-c432-4d1a-b120-58fd7113609f; Name: New Customer 1; City: Brooklyn
 Id: 34e9873a-94c8-4720-9146-d63fb7840fad; Name: New Customer 1; City: Brooklyn
```

18. DocumentDB SQL – JavaScript Integration

These days JavaScript is everywhere, and not just in browsers. DocumentDB embraces JavaScript as a sort of modern day T-SQL and supports the transactional execution of JavaScript logic natively, right inside the database engine. DocumentDB provides a programming model for executing JavaScript-based application logic directly on the collections in terms of stored procedures and triggers.

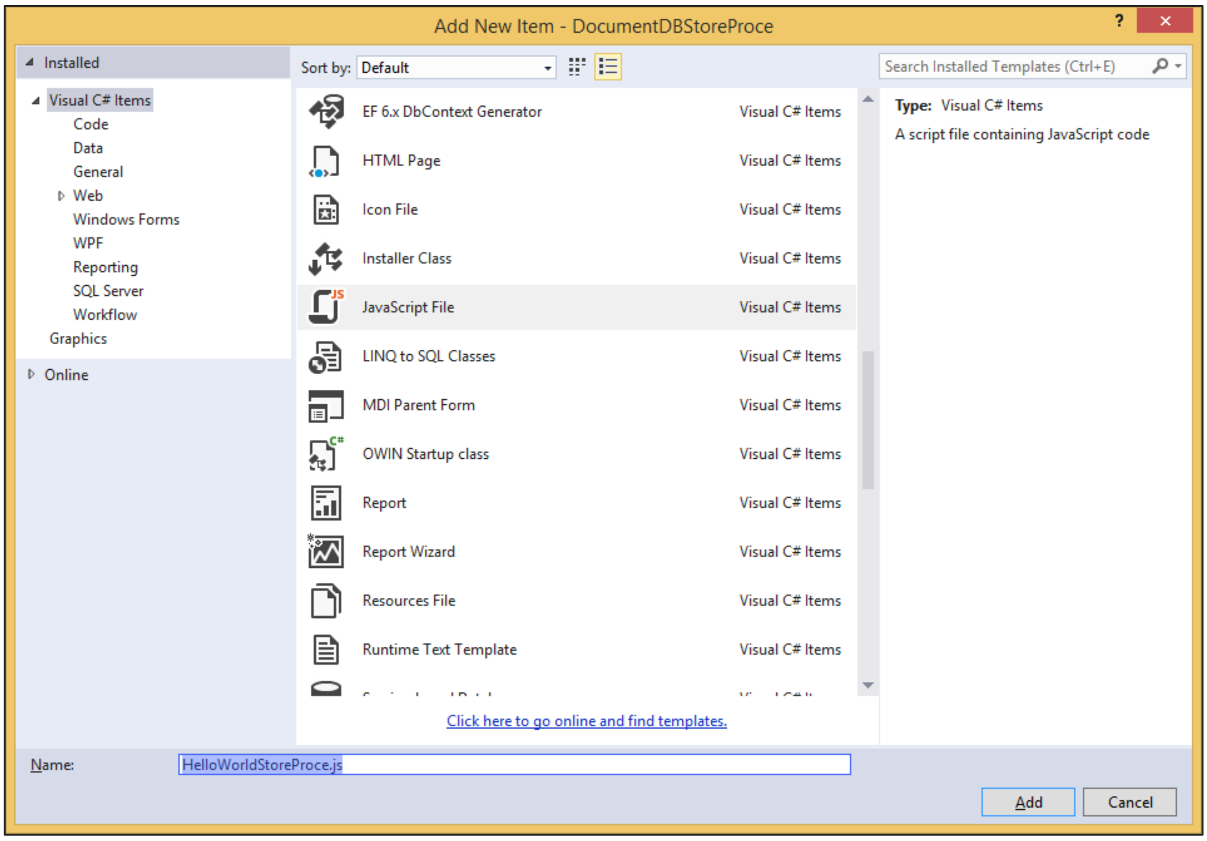
Let’s take a look at an example where we create a simple store procedure. Following are the steps:

Step 1: Create a new console applications.

Step 2: Add in the .NET SDK from NuGet. We are using the .NET SDK here, which means that we'll be writing some C# code to create, execute, and then delete our stored procedure, but the stored procedure itself gets written in JavaScript.

Step 3: Right-click on the project in Solution explorer.

Step 4: Add a new JavaScript file for the stored procedure and call it HelloWorldStoreProce.js



Type: Visual C# Items

A script file containing JavaScript code

Name: HelloWorldStoreProce.js

Every stored procedure is just a JavaScript function so we'll create a new function and naturally we'll also name this function **HelloWorldStoreProce**. It doesn't matter if we give the function a name at all. DocumentDB will only refer to this stored procedure by the Id that we provide when we create it.

```
function HelloWorldStoreProce() {
    var context = getContext();
    var response = context.getResponse();
    response.setBody('Hello, and welcome to DocumentDB!');
}
```

All the stored procedure does is obtain the response object from the context and call its **setBody** method to return a string to the caller. In C# code, we will create the stored procedure, execute it, and then delete it.

Stored procedures are scoped per collection, therefore we will need the SelfLink of the collection to create the stored procedure.

Step 5: First query for the **myfirstdb** database and then for the **MyCollection** collection.

Creating a stored procedure is just like creating any other resource in DocumentDB.

```
private async static Task SimpleStoredProcDemo()
{

    var endpoint = "https://azuredocdbdemo.documents.azure.com:443/";
    var masterKey =
"BBhjI0gxdVPdDbS4diTjdloJq7Fp4L5RO/StTt6UtEufDM78qM2CtBZWbyVwFPSJIm8AcfDu2O+AfV
T+TYUnBQ==";

    using (var client = new DocumentClient(new Uri(endpoint), masterKey))
    {
        // Get database
        Database database = client
            .CreateDatabaseQuery("SELECT * FROM c WHERE c.id = 'myfirstdb'")
            .AsEnumerable()
            .First();

        // Get collection
        DocumentCollection collection = client
            .CreateDocumentCollectionQuery(database.CollectionsLink, "SELECT *
FROM c WHERE c.id = 'MyCollection'")
            .AsEnumerable()
            .First();

        // Create stored procedure
        var sprocBody = File.ReadAllText(@"..\..\HelloWorldStoreProce.js");
        var sprocDefinition = new StoredProcedure
        {
            Id = "HelloWorldStoreProce",
            Body = sprocBody
        };


        StoredProcedure sproc = await
client.CreateStoredProcedureAsync(collection.SelfLink, sprocDefinition);
        Console.WriteLine("Created stored procedure {0} ({1})", sproc.Id,
sproc.ResourceId);


        // Execute stored procedure
        var result = await
client.ExecuteStoredProcedureAsync<string>(sproc.SelfLink);
        Console.WriteLine("Executed stored procedure; response = {0}",
result.Response);


        // Delete stored procedure
        await client.DeleteStoredProcedureAsync(sproc.SelfLink);
        Console.WriteLine("Deleted stored procedure {0} ({1})", sproc.Id,
sproc.ResourceId);
    }


}
```

Step 6: First create a definition object with the Id for the new resource and then call one of the Create methods on the **DocumentClient** object. In the case of a stored procedure, the definition includes the Id and the actual JavaScript code that you want to ship over to the server.

Step 7: Call **File.ReadAllText** to extract the stored procedure code out of the JS file.

Step 8: Assign the stored procedure code to the body property of the definition object.

As far as DocumentDB is concerned, the Id we specify here, in the definition, is the name of the stored procedure, regardless of what we actually name the JavaScript function.

Nevertheless when creating stored procedures and other server-side objects, it is recommended that we name JavaScript functions and that those function names do match the Id that we have set in the definition for DocumentDB.

Step 9: Call **CreateStoredProcedureAsync**, passing in the **SelfLink** for the **MyCollection** collection and the stored procedure definition. This creates the stored procedure and **ResourceId** that DocumentDB assigned to it.

Step 10: Call the stored procedure. **ExecuteStoredProcedureAsync** takes a type parameter that you set to the expected data type of the value returned by the stored procedure, which you can specify simply as an object if you want a dynamic object returned. That is an object whose properties will be bound at run-time.

In this example we know that our stored procedure is just returning a string and so we call **ExecuteStoredProcedureAsync<string>**.

Following is the complete implementation of Program.cs file.

```
using Microsoft.Azure.Documents;
using Microsoft.Azure.Documents.Client;
using Microsoft.Azure.Documents.Linq;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DocumentDBStoreProce
{
    class Program
    {
        private static void Main(string[] args)
        {
            Task.Run(async () =>
            {
                await SimpleStoredProcDemo();
            }).Wait();
        }

        private async static Task SimpleStoredProcDemo()
        {

            var endpoint = "https://azuredocdbdemo.documents.azure.com:443/";
            var masterKey =
"BBhjI0gxdVPdDbS4diTjdloJq7Fp4L5RO/StTt6UtEufDM78qM2CtBZWbyVwFPSJIm8AcfDu2O+AfV
T+TYUnBQ==";

            using (var client = new DocumentClient(new Uri(endpoint), masterKey))
            {
                // Get database
                Database database = client
                    .CreateDatabaseQuery("SELECT * FROM c WHERE c.id = 'myfirstdb'")
```

```
                     .AsEnumerable()
                     .First();

                 // Get collection
                 DocumentCollection collection = client
                     .CreateDocumentCollectionQuery(database.CollectionsLink,
"SELECT * FROM c WHERE c.id = 'MyCollection'")
                     .AsEnumerable()
                     .First();

                 // Create stored procedure
                 var sprocBody = File.ReadAllText(@"..\..\HelloWorldStoreProce.js");
                 var sprocDefinition = new StoredProcedure
                 {
                     Id = "HelloWorldStoreProce",
                     Body = sprocBody
                 };

                 StoredProcedure sproc = await
client.CreateStoredProcedureAsync(collection.SelfLink, sprocDefinition);
                 Console.WriteLine("Created stored procedure {0} ({1})",
sproc.Id, sproc.ResourceId);

                 // Execute stored procedure
                 var result = await
client.ExecuteStoredProcedureAsync<string>(sproc.SelfLink);
                 Console.WriteLine("Executed stored procedure; response = {0}",
result.Response);

                 // Delete stored procedure
                 await client.DeleteStoredProcedureAsync(sproc.SelfLink);
                 Console.WriteLine("Deleted stored procedure {0} ({1})",
sproc.Id, sproc.ResourceId);
             }
         }
     }
}
```

When the above code is executed, it produces the following output.

```
Created stored procedure HelloWorldStoreProce (Ic8LAMEUVgACAAAAAAAAAgA==)
```

```
Executed stored procedure; response = Hello, and welcome to DocumentDB!
```

As seen in the above output, the response property has the "Hello, and welcome to DocumentDB!" returned by our stored procedure.

19. DocumentDB SQL – User-defined Functions

DocumentDB SQL provides support for User-Defined Functions (UDFs). UDFs are just another kind of JavaScript functions you can write and these work pretty much as you'd expect. You can create UDFs to extend the query language with custom business logic that you can reference in your queries.

The DocumentDB SQL syntax is extended to support custom application logic using these UDFs. UDFs can be registered with DocumentDB and then be referenced as part of a SQL query.

Let’s consider the following three documents for this example.

AndersenFamily document is as follows.

```
{
     "id": "AndersenFamily",
     "lastName": "Andersen",
     "parents": [
            { "firstName": "Thomas", "relationship":  "father" },
            { "firstName": "Mary Kay", "relationship":  "mother" }
     ],
     "children": [
            {
                   "firstName": "Henriette Thaulow",
                   "gender": "female",
                   "grade": 5,
                   "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
            }
     ],
     "location": { "state": "WA", "county": "King", "city": "Seattle" },
     "isRegistered": true
}
```

SmithFamily document is as follows.

```
{
     "id": "SmithFamily",
     "parents": [
            { "familyName": "Smith", "givenName": "James" },
            { "familyName": "Curtis", "givenName": "Helen" }
```

```
	],
	"children": [
		{
			"givenName": "Michelle",
			"gender": "female",
			"grade": 1
		},
		{
			"givenName": "John",
			"gender": "male",
			"grade": 7,
			"pets": [
				{ "givenName": "Tweetie", "type": "Bird" }
			]
		}
	],
	"location": {
		"state": "NY",
		"county": "Queens",
		"city": "Forest Hills"
	},
	"isRegistered": true
}
```

WakefieldFamily document is as follows.

```
{
	"id": "WakefieldFamily",
	"parents": [
		{ "familyName": "Wakefield", "givenName": "Robin" },
		{ "familyName": "Miller", "givenName": "Ben" }
	],
	"children": [
		{
			"familyName": "Merriam",
			"givenName": "Jesse",
			"gender": "female",
			"grade": 6,
			"pets": [
                        { "givenName": "Charlie Brown", "type": "Dog" },
                { "givenName": "Tiger", "type": "Cat" },
                { "givenName": "Princess", "type": "Cat" }
                  ]
            },
            {
                  "familyName": "Miller",
                  "givenName": "Lisa",
                  "gender": "female",
                  "grade": 3,
                  "pets": [
                        { "givenName": "Jake", "type": "Snake" }
                  ]
            }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```

Let’s take a look at an example where we will create some simple UDFs.

Following is the implementation of **CreateUserDefinedFunctions**.

```
private async static Task CreateUserDefinedFunctions(DocumentClient client)
{
      Console.WriteLine();
      Console.WriteLine("**** Create User Defined Functions ****");
      Console.WriteLine();

      await CreateUserDefinedFunction(client, "udfRegEx");

}
```

We have a udfRegEx, and in CreateUserDefinedFunction we get its JavaScript code from our local file. We construct the definition object for the new UDF, and call

CreateUserDefinedFunctionAsync with the collection's SelfLink and the udfDefinition object as shown in the following code.

```
private async static Task<UserDefinedFunction>
CreateUserDefinedFunction(DocumentClient client, string udfId)
{
     var udfBody = File.ReadAllText(@"..\..\Server\" + udfId + ".js");
     var udfDefinition = new UserDefinedFunction
     {
            Id = udfId,
            Body = udfBody
     };

     var result = await
client.CreateUserDefinedFunctionAsync(_collection.SelfLink, udfDefinition);
     var udf = result.Resource;
     Console.WriteLine("Created user defined function {0}; RID: {1}", udf.Id,
udf.ResourceId);

     return udf;
}
```

We get back the new UDF from the resource property of the result and return it back up to the caller. To display the existing UDF, following is the implementation of **ViewUserDefinedFunctions**. We call **CreateUserDefinedFunctionQuery** and loop through them as usual.

```
private static void ViewUserDefinedFunctions(DocumentClient client)
{
     Console.WriteLine();
     Console.WriteLine("**** View UDFs ****");
     Console.WriteLine();

     var udfs = client

     .CreateUserDefinedFunctionQuery(_collection.UserDefinedFunctionsLink)
            .ToList();

     foreach (var udf in udfs)
     {
            Console.WriteLine("User defined function {0}; RID: {1}", udf.Id,
udf.ResourceId);
     }
}
```

DocumentDB SQL doesn't provide built-in functions to search for substrings or for regular expressions, hence the following little one-liner fills that gap which is a JavaScript function.

```
function udfRegEx(input, regex) {
     return input.match(regex);
}
```

Given the input string in the first parameter, use JavaScript's built-in regular expression support passing in the pattern matching string in the second parameter into **.match**. We can run a substring query to find all stores with the word Andersen in their **lastName** property.

```
private static void Execute_udfRegEx(DocumentClient client)
{
     var sql = "SELECT c.name FROM c WHERE udf.udfRegEx(c.lastName, 'Andersen')
!= null";

     Console.WriteLine();
     Console.WriteLine("Querying for Andersen");
     var documents = client.CreateDocumentQuery(_collection.SelfLink,
sql).ToList();

     Console.WriteLine("Found {0} Andersen:", documents.Count);
     foreach (var document in documents)
     {
            Console.WriteLine("Id: {0}, Name: {1}", document.id,
document.lastName);
     }
}
```

Note that we must qualify every UDF reference with the prefix **udf**. We just passed the SQL along to **CreateDocumentQuery** like any ordinary query. Finally, let's call the above queries from the **CreateDocumentClient** task.

```
private static async Task CreateDocumentClient()
{
    // Create a new instance of the DocumentClient
    using (var client = new DocumentClient(new Uri(EndpointUrl),
AuthorizationKey))
    {
        database = client.CreateDatabaseQuery("SELECT * FROM c WHERE c.id =
'myfirstdb'").AsEnumerable().First();
```

```
        collection =
client.CreateDocumentCollectionQuery(database.CollectionsLink, "SELECT * FROM c
WHERE c.id = 'Families'").AsEnumerable().First();


        await CreateUserDefinedFunctions(client);


      ViewUserDefinedFunctions(client);


      Execute_udfRegEx(client);
    }
}
```

When the above code is executed, it produces the following output.

```
**** Create User Defined Functions ****


Created user defined function udfRegEx; RID: kV5oANVXnwAlAAAAAAAAYA==


**** View UDFs ****


User defined function udfRegEx; RID: kV5oANVXnwAlAAAAAAAAYA==


Querying for Andersen
Found 1 Andersen:
 Id: AndersenFamily, Name: Andersen
```

20. DocumentDB SQL – Composite SQL Queries

Composite Query enables you to combine data from existing queries and then apply filters, aggregates, and so on before presenting the report results, which show the combined data set. Composite Query retrieves multiple levels of related information on existing queries and presents the combined data as a single and flattened query result.

Using Composite Query, you also have the option to:

- Select the SQL pruning option to remove tables and fields that are not needed based on users' attribute selections.

- Set the ORDER BY and GROUP BY clauses.

- Set the WHERE clause as a filter over the result set of a composite query.

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

Let's consider the following documents for this example.

AndersenFamily document is as follows.

```
{
      "id": "AndersenFamily",
      "lastName": "Andersen",
      "parents": [
             { "firstName": "Thomas", "relationship":  "father" },
             { "firstName": "Mary Kay", "relationship":  "mother" }
      ],
      "children": [
             {
                    "firstName": "Henriette Thaulow",
                    "gender": "female",
                    "grade": 5,
                    "pets": [ { "givenName": "Fluffy", "type":  "Rabbit" } ]
             }
      ],
      "location": { "state": "WA", "county": "King", "city": "Seattle" },
      "isRegistered": true
}
```

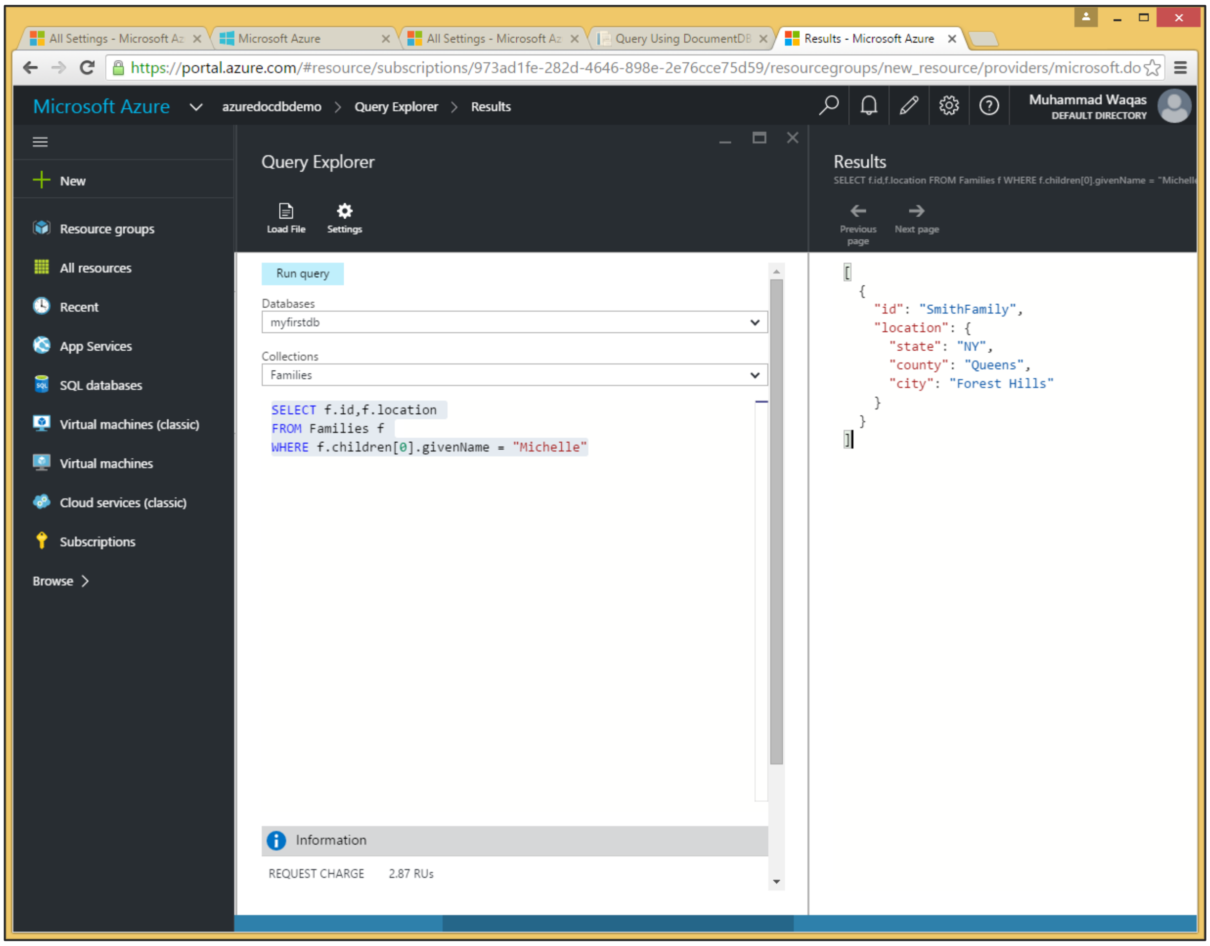
SmithFamily document is as follows.

```
{
	"id": "SmithFamily",
	"parents": [
		{ "familyName": "Smith", "givenName": "James" },
		{ "familyName": "Curtis", "givenName": "Helen" }
	],
	"children": [
		{
			"givenName": "Michelle",
			"gender": "female",
			"grade": 1
		},
		{
			"givenName": "John",
			"gender": "male",
			"grade": 7,
			"pets": [
				{ "givenName": "Tweetie", "type": "Bird" }
			]
		}
	],
	"location": {
		"state": "NY",
		"county": "Queens",
		"city": "Forest Hills"
	},
	"isRegistered": true
}
```

WakefieldFamily document is as follows.

```
{
	"id": "WakefieldFamily",
	"parents": [
		{ "familyName": "Wakefield", "givenName": "Robin" },
		{ "familyName": "Miller", "givenName": "Ben" }
	],
      "children": [
             {
                    "familyName": "Merriam",
                    "givenName": "Jesse",
                    "gender": "female",
                    "grade": 6,
                    "pets": [
                           { "givenName": "Charlie Brown", "type": "Dog" },
                  { "givenName": "Tiger", "type": "Cat" },
                  { "givenName": "Princess", "type": "Cat" }
                    ]
             },
             {
                    "familyName": "Miller",
                    "givenName": "Lisa",
                    "gender": "female",
                    "grade": 3,
                    "pets": [
                           { "givenName": "Jake", "type": "Snake" }
                    ]
             }
      ],
      "location": { "state": "NY", "county": "Manhattan", "city": "NY" },
      "isRegistered": false
}
```

Let’s take a look at an example of concatenated query.


All Settings - Microsoft Az
Microsoft Azure
All Settings - Microsoft Az
Query Using DocumentDB
Results - Microsoft Azure
https://portal.azure.com/#resource/subscriptions/973ad1fe-282d-4646-898e-2e76cce75d59/resourcegroups/new_resource/providers/microsoft.do
Microsoft Azure
azuredocdbdemo > Query Explorer > Results
Muhammad Waqas
DEFAULT DIRECTORY
New
Resource groups
All resources
Recent
App Services
SQL databases
Virtual machines (classic)
Virtual machines
Cloud services (classic)
Subscriptions
Browse >
Query Explorer
Load File
Settings
Run query
Results
Previous page
Next page

Databases: myfirstdb

Collections: Families

```
SELECT f.id,f.location
FROM Families f
WHERE f.children[0].givenName = "Michelle"
```

```
[
  {
    "id": "SmithFamily",
    "location": {
      "state": "NY",
      "county": "Queens",
      "city": "Forest Hills"
    }
  }
]
```

Information

REQUEST CHARGE 2.87 RUs

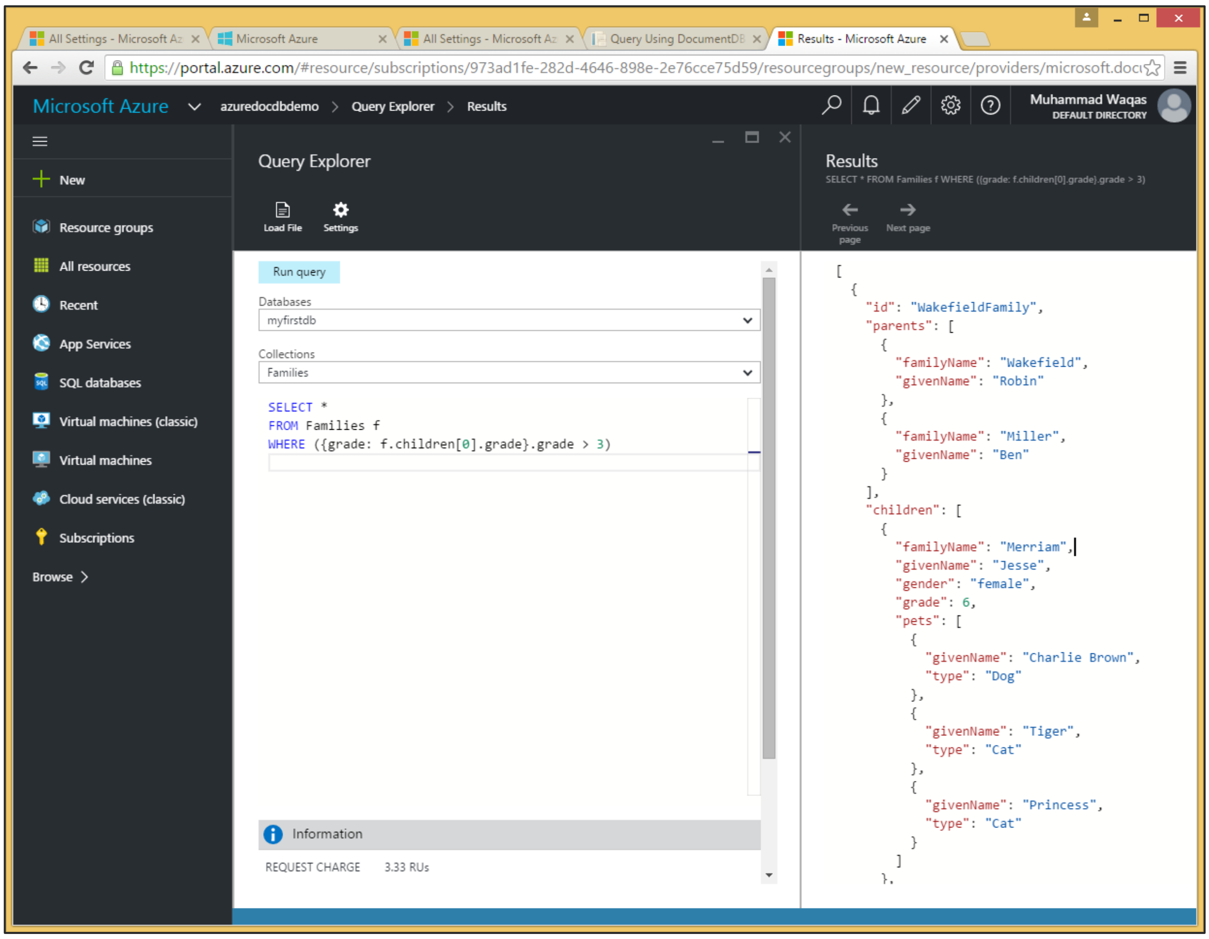
Following is the query which will retrieve the id and location of the family where the first child **givenName** is Michelle.

```
SELECT f.id,f.location
FROM Families f
WHERE f.children[0].givenName = "Michelle"
```

When the above query is executed, it produces the following output.

```
[  {
    "id": "SmithFamily",
    "location": {
      "state": "NY",
      "county": "Queens",
      "city": "Forest Hills"
    }  }]
```

Let’s consider another example of concatenated query.



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

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

```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
      },
```

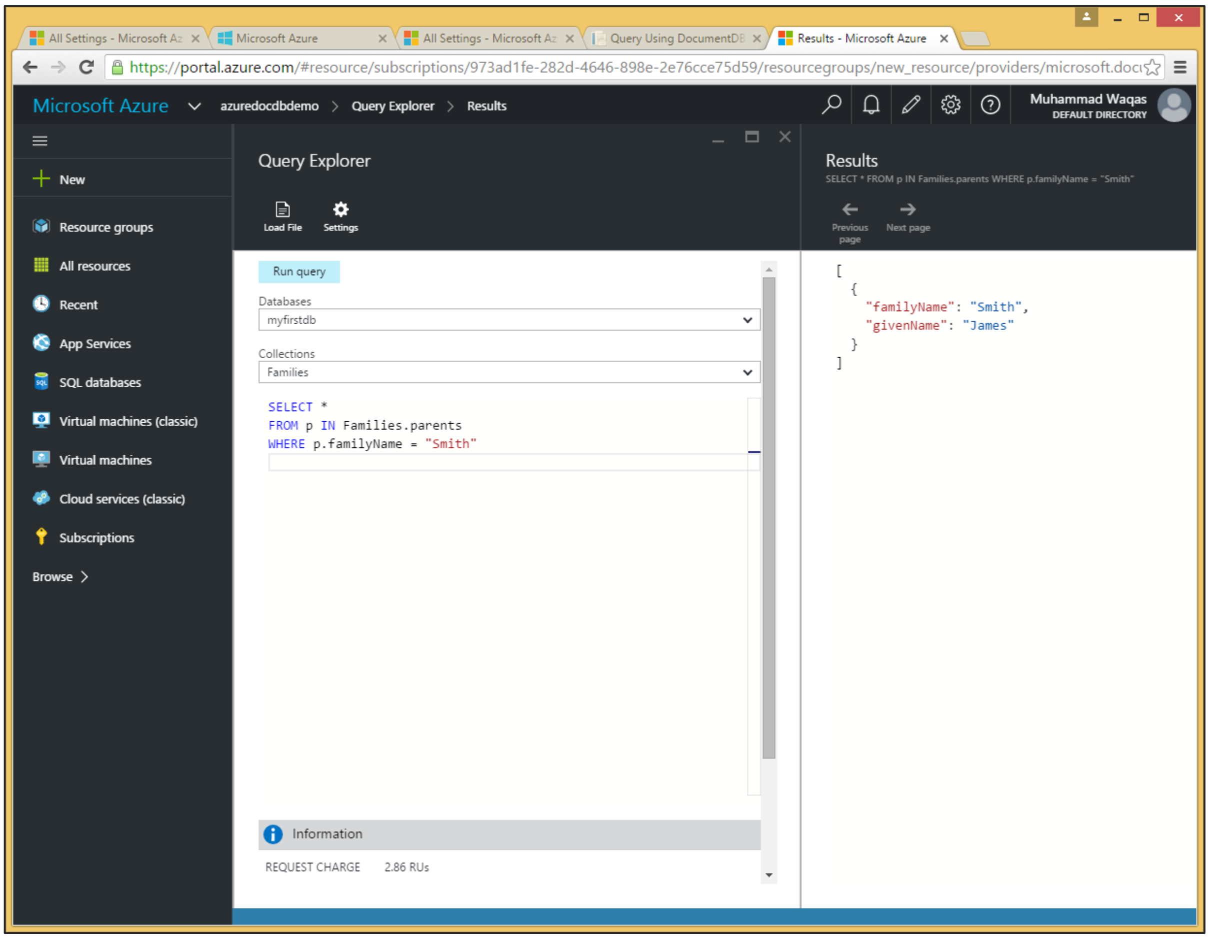
Following is the query which will return all the documents where the first child grade greater than 3.

```
SELECT *

FROM Families f

WHERE ({grade: f.children[0].grade}.grade > 3)
```

When the above query is executed, it produces the following output.

```
[
  {
    "id": "WakefieldFamily",
    "parents": [
      {
        "familyName": "Wakefield",
        "givenName": "Robin"
      },
      {
        "familyName": "Miller",
        "givenName": "Ben"
      }
    ],
    "children": [
      {
        "familyName": "Merriam",
        "givenName": "Jesse",
        "gender": "female",
        "grade": 6,
        "pets": [
          {
            "givenName": "Charlie Brown",
            "type": "Dog"
          },
          {
            "givenName": "Tiger",
            "type": "Cat"
          },
          {
            "givenName": "Princess",
            "type": "Cat"
          }
        ]
      },
      {
        "familyName": "Miller",
        "givenName": "Lisa",
        "gender": "female",
        "grade": 3,
        "pets": [
          {
            "givenName": "Jake",
            "type": "Snake"
          }
        ]
      }
    ],
    "location": {
      "state": "NY",
      "county": "Manhattan",
      "city": "NY"
    },
    "isRegistered": false,
    "_rid": "Ic8LAJFujgECAAAAAAAAAA==",
    "_ts": 1450541623,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgECAAAAAAAAAA==/",
    "_etag": "\"00000500-0000-0000-0000-567582370000\"",
    "_attachments": "attachments/"
  },
  {
    "id": "AndersenFamily",
    "lastName": "Andersen",
    "parents": [
      {
        "firstName": "Thomas",
        "relationship": "father"
      },
      {
        "firstName": "Mary Kay",
        "relationship": "mother"
      }
    ],
    "children": [
      {
        "firstName": "Henriette Thaulow",
        "gender": "female",
        "grade": 5,
        "pets": [
          {
            "givenName": "Fluffy",
            "type": "Rabbit"
          }
        ]
      }
    ],
    "location": {
      "state": "WA",
      "county": "King",
      "city": "Seattle"
    },
    "isRegistered": true,
    "_rid": "Ic8LAJFujgEEAAAAAAAAAA==",
    "_ts": 1450541624,
    "_self": "dbs/Ic8LAA==/colls/Ic8LAJFujgE=/docs/Ic8LAJFujgEEAAAAAAAAAA==/",
    "_etag": "\"00000700-0000-0000-0000-567582380000\"",
    "_attachments": "attachments/"
  }
]
```

Let's take a look at an **example** of nested queries.



Databases: myfirstdb

Collections: Families

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

Results

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

```
[
  {
    "familyName": "Smith",
    "givenName": "James"
  }
]
```

REQUEST CHARGE 2.86 RUs

Following is the query which will iterate all the parents and then return the document where **familyName** is Smith.

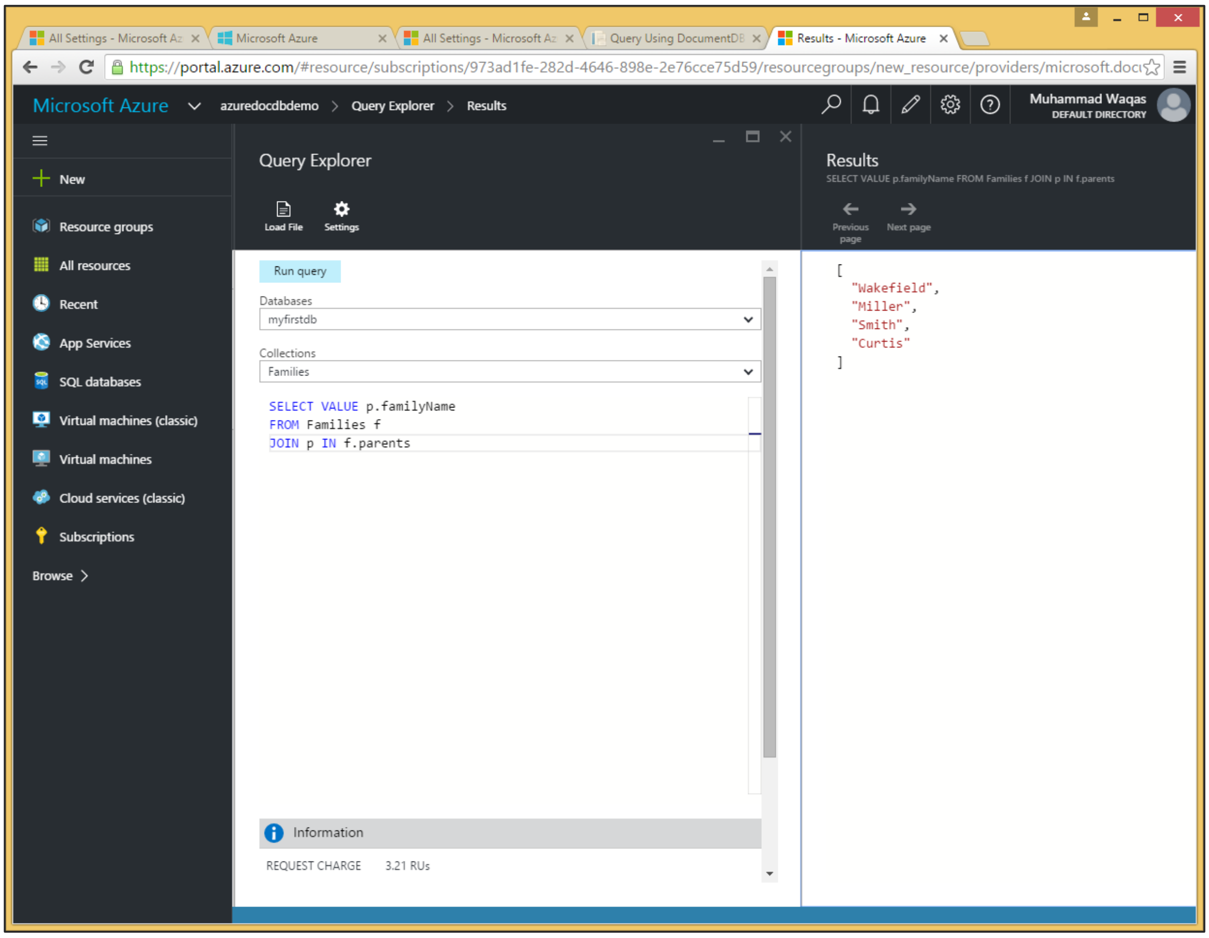
```
SELECT *
```

```
FROM p IN Families.parents
WHERE p.familyName = "Smith"
```

When the above query is executed, it produces the following output.

```
[
  {
    "familyName": "Smith",
    "givenName": "James"
  }
]
```

Let’s consider **another example** of nested query.



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

```
[
  "Wakefield",
  "Miller",
  "Smith",
  "Curtis"
]
```

Following is the query which will return all the **familyName**.

```
SELECT VALUE p.familyName
```

```
FROM Families f
JOIN p IN f.parents
```

When the above query is executed, it produces he following output.

```
[
  "Wakefield",
  "Miller",
  "Smith",
  "Curtis"
]
```