Microsoft

Execute queries in Azure Cosmos DB for NoSQL

Agenda

- Query the Azure Cosmos DB for NoSQL
- Author complex queries with the Azure Cosmos DB for NoSQL

Query the Azure Cosmos DB for NoSQL

Create queries with SQL

Azure Cosmos DB for NoSQL uses (SQL) syntax to perform queries over semi-structured data.

A few examples of Azure Cosmos DB for NoSQL SQL queries.

```
SELECT * FROM products
```

```
SELECT products.id,
    products.name,
    products.price,
    products.categoryName FROM products
```

```
SELECT p.name,
    p.price
FROM p
```

```
SELECT ReferenceContainerHoweverYouLike.name,
 ReferenceContainerHoweverYouLike.price
FROM ReferenceContainerHoweverYouLike
```

```
SELECT p.name,
    p.categoryName,
    p.price
FROM products p
WHERE p.price >= 50
    AND p.price <= 100
```

Project query results

Azure Cosmos DB for NoSQL extends SQL to manipulate JSON results.

AS

```
SELECT p.categoryName AS category
FROM products p
```

DISTINCT

```
SELECT DISTINCT p.categoryName
FROM products p
```

```
public class CategoryReader {
    public string CategoryName { get; set; }
}
// Developers read this as List<CategoryReader>
```

VALUE

```
SELECT VALUE p.categoryName
FROM products p
```

```
// Developers read this as List<string>
```

Child properties

```
// Suppose we have the following ProductAdvertisement class definition.
public class ProductAdvertisement {
    public string Name { get; set; }
    public string Category { get; set; }
    public class ScannerData { public decimal Price { get; set; } }
}
```

```
// scannerData expression added to return the expected class format.
SELECT p.name,
    p.categoryName AS category,
    { "price": p.price } AS scannerData
FROM products p
WHERE p.price >= 50 AND p.price <= 100
```

```
// The query will return this JSON.
{
    "name": "LL Bottom Bracket",
    "category": "Components, Bottom Brackets",
    "scannerData": { "price": 53.99 }
}
```

Implement type-checking in queries

NoSQL is schema-less, the responsibility for type checking will often fall on your queries.

IS_DEFINED

```
// Let's assume this is a document on the Product container.
{
    "id": "6374995F-9A78-43CD-AE0D-5F6041078140",
    "categoryid": "3E4CEACD-D007-46EB-82D7-31F6141752B2",
    "sku": "FR-R38R-60",
    "name": "LL Road Frame - Red, 60",
    "price": 337.22
}
```

```
-- Note how in the previous document there are no tags properties
SELECT IS_DEFINED(p.tags) AS tags_exist
FROM products p
```

```
// This query returns.
[ { "tags_exist": false } ]
```

Other type-checking functions

- IS_BOOLEAN
- IS_OBJECT

IS_ARRAY, IS_NULL, IS_STRING, IS_NUMBER

```
// Let's assume this is a document on the Product container.
{
    "id": "6374995F-9A78-43CD-AE0D-5F6041078140",
    "categoryid": "3E4CEACD-D007-46EB-82D7-31F6141752B2",
    "sku": "FR-R38R-60",
    "name": "LL Road Frame - Red, 60",
    "price": 337.22, "tags": "fun, sporty, rad" }
```

```
-- IS_ARRAY
SELECT IS_ARRAY(p.tags) AS tags_is_array
FROM products p

-- IS_NULL
SELECT IS_NULL(p.tags) AS tags_is_null
FROM products p

-- IS_NUMBER
SELECT p.id, p.price, (p.price * 1.25) AS priceWithTax
FROM products p
WHERE IS_NUMBER(p.price)

-- IS_STRING, returns false
SELECT p.id, p.price
FROM products p
WHERE IS_STRING(p.price)
```

Use built-in functions

SQL for the Azure Cosmos DB for NoSQL ships with built-in functions for common tasks in a query.

Here are some examples of these functions:

CONCAT

```
SELECT VALUE CONCAT(p.name, ' | ', p.categoryName)
FROM products p
```

GETCURRENTDATETIME

```
SELECT *
FROM products p
WHERE p.retirementDate >= GetCurrentDateTime()
```

LOWER

```
SELECT VALUE LOWER(p.sku)
FROM products p
```

CONTAINS

```
SELECT p.CompanyName
FROM companies p
WHERE CONTAINS(p.CompanyName, "contoso", true)
```

RTRIM, LTRIM, LEFT, RIGHT

```
SELECT VALUE LTRIM(RTRIM(p.sku))
FROM products p

SELECT VALUE LEFT(p.fullname, 10)
FROM Customers c
```

ARRAY_CONTAINS

```
SELECT p.ProductName
FROM products p
WHERE ARRAY_CONTAINS(p.tags, "wheel", true)
```

Execute queries in the SDK

The *Microsoft.Azure.Cosmos.Container* class in the SDK has built in classes to manipulate queries.

```
// Suppose we want to run the query SELECT * FROM products p.

// Let's define a class for our products.
public class Product
{
    public string id { get; set; }
    public string name { get; set; }
    public string price { get; set; }
}

// Let's create our query definition.
QueryDefinition query = new ("SELECT * FROM products p");

// Let's finally execute our query. We will run it inside a foreach loop in case we get multiple documents back

await foreach (Product product in container.GetItemQueryIterator<Product>(query))
{
    Console.WriteLine($"[{product.id}]\t{product.name,35}\t{product.price,15:C}");
}
```

Lab – Execute a query with the Azure Cosmos DB for NoSQL SDK

- Prepare your development environment
- Create an Azure Cosmos DB for NoSQL account
- Seed the Azure Cosmos DB for NoSQL account with data
- Iterate over the results of a SQL query using the SDK

Author complex queries with the Azure Cosmos DB for NoSQL

Create a transactional batch with the SDK

Unlike a JOIN in a relational database, a JOIN in Azure Cosmos DB for NoSQL scope is a single item only. A JOIN creates a cross-product between different sections of a single item.

```
// Suppose this is a document in our product container.
{
    "id": "80D3630F-B661-4FD6-A296-CD03BB7A4A0C",
    "categoryId": "629A8F3C-CFB0-4347-8DCC-505A4789876B",
    "categoryName": "Clothing, Vests",
    "sku": "VE-C304-L",
    "name": "Classic Vest, L",
    "description": "A worn brown classic",
    "price": 32.4,
    "tags": [
              {
                "id": "2CE9DADE-DCAC-436C-9D69-B7C886A01B77",
                "name": "apparel", "class": "group"
              },
              {
                "id": "CA170AAD-A5F6-42FF-B115-146FADD87298",
                "name": "worn", "class": "trade-in"
              },
              {
                "id": "CA170AAD-A5F6-42FF-B115-146FADD87298",
                "name": "no-damaged", "class": "trade-in"
              }
            ]
}
```

```
-- Return the tag name for all tags embedded in the document
SELECT p.id, p.name, t.name AS tag
FROM products p
JOIN t IN p.tags
```

```
// JSON returned by this query for the document on the left.
[
    {
        "id": "80D3630F-B661-4FD6-A296-CD03BB7A4A0C",
        "name": "Classic Vest, L",
        "tag": "apparel"
    },
    {
         "id": "80D3630F-B661-4FD6-A296-CD03BB7A4A0C",
         "name": "Classic Vest, L",
         "tag": "worn"
    },
    {
         "id": "80D3630F-B661-4FD6-A296-CD03BB7A4A0C",
         "name": "Classic Vest, L",
         "tag": "no-damaged"
     }
]
```

Implement correlated subqueries

We can optimize JOIN expressions further by writing subqueries to filter the number of array items we want to include in the cross-product set.

```
// Suppose this is a document in our product container.
{
    "id": "80D3630F-B661-4FD6-A296-CD03BB7A4A0C",
    "categoryId": "629A8F3C-CFB0-4347-8DCC-505A4789876B",
    "categoryName": "Clothing, Vests",
    "sku": "VE-C304-L",
    "name": "Classic Vest, L",
    "description": "A worn brown classic",
    "price": 32.4,
    "tags": [
              {
                "id": "2CE9DADE-DCAC-436C-9D69-B7C886A01B77",
                "name": "apparel", "class": "group"
              },
              {
                "id": "CA170AAD-A5F6-42FF-B115-146FADD87298",
                "name": "worn", "class": "trade-in"
              },
              {
                "id": "CA170AAD-A5F6-42FF-B115-146FADD87298",
                "name": "no-damaged", "class": "trade-in"
              }
            ]
}
```

```
-- Return the tag name for trade-in tags embedded in the document
SELECT p.id,
    p.name,
    t.name AS tag
FROM products p
JOIN (SELECT VALUE t
      FROM t IN p.tags
      WHERE t.class = 'trade-in') AS t
```

```
// JSON returned by this query for the document on the left.
[
    {
        "id": "80D3630F-B661-4FD6-A296-CD03BB7A4A0C",
        "name": "Classic Vest, L",
        "tag": "worn"
    },
    {
        "id": "80D3630F-B661-4FD6-A296-CD03BB7A4A0C",
        "name": "Classic Vest, L",
        "tag": "no-damaged"
     }
]
```

Implement variables in queries

Using the *QueryDefinition* class, we can add query parameters.

```
// Suppose we want to run the query below from the SDK, but we would like to send the condition values as parameters
// SELECT p.name, t.name AS tag FROM products p JOIN t IN p.tags WHERE p.price > 500 AND p.price <= 1000.

// Let's define a class for our products.
public class Product
{
    public string id { get; set; }
    public string name { get; set; }
    public string price { get; set; }
}

// Let's create our query definition.
string sql = "SELECT p.name, t.name AS tag FROM products p JOIN t IN p.tags WHERE p.price >= @lower AND p.price <= @upper"

QueryDefinition query = new (sql)
    .WithParameter("@lower", 500)
    .WithParameter("@upper", 1000);

// Let's finally execute our query. We will run it inside a foreach loop in case we get multiple documents back

await foreach (Product product in container.GetItemQueryIterator<Product>(query))
{
    Console.WriteLine($"[{product.id}]\t{product.name,35}\t{product.price,15:C}");
}
```

Paginate query results

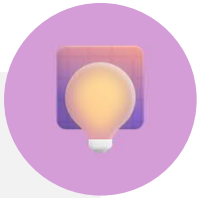
The *Microsoft.Azure.Cosmos.Container* class supports asynchronous streams to iterate over multiple pages of results.

```
// Suppose we want to run the query below from the SDK, but we would like to return 100 documents at a time
// SELECT * FROM products p WHERE p.price > 500.

// Let's define a class for our products.
public class Product
{
    public string id { get; set; }
    public string name { get; set; }
    public string price { get; set; }
}

// Let's create our query definition.
string sql = "SELECT * FROM products WHERE p.price > 500";

QueryDefinition query = new (sql)
QueryRequestOptions options = new() { MaxItemCount = 100 };

// Let's finally execute our query. We will run it inside a foreach and a while loop in case we get multiple documents back
FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOptions: options);

while(iterator.HasMoreResults)
{
    foreach(Product product in await iterator.ReadNextAsync())
    { // Handle individual items }
}
```

Lab – Paginate cross-product query results with the Azure Cosmos DB for NoSQL SDK

- Prepare your development environment
- Create an Azure Cosmos DB for NoSQL account
- Seed the Azure Cosmos DB for NoSQL account with data
- Paginate through small result sets of a SQL query using the SDK

Review



1. Which SQL keyword is used to flatten your query results into an array of a specific field value for each item?

   - ❑ FROM.
   - ❑ DISTINCT.
   - ☑ VALUE.

2. You have a container named products. What must the data source be named after the *FROM* keyword in the query?

   - ❑ p.
   - ❑ product.
   - ☑ Any of these.

3. Which method of the Microsoft.Azure.Cosmos.Container class takes in a SQL query as a string parameter and returns an iterator that can be used to iterate over the query results as deserialized C# objects?

   - ❑ GetItemQueryStreamIterator<>
   - ☑ GetItemQueryIterator<>
   - ❑ GetItemLinqQueryable<>