

# LAB02 : Querying and Indexing in Azure Cosmos DB

## A-Querying in Azure Cosmos DB

Azure Cosmos DB SQL API accounts provide support for querying items using the Structured Query Language (SQL), one of the most familiar and popular query languages, as a JSON query language. In this lab, you will explore how to use these rich query capabilities directly through the Azure Portal. No separate tools or client side code are required.

If this is your first lab and you have not already completed the setup for the lab content see the instructions for [Account Setup](#) before starting this lab.

### Query Overview

Querying JSON with SQL allows Azure Cosmos DB to combine the advantages of a legacy relational databases with a NoSQL database. You can use many rich query capabilities such as sub-queries or aggregation functions but still retain the many advantages of modeling data in a NoSQL database.

Azure Cosmos DB supports strict JSON items only. The type system and expressions are restricted to deal only with JSON types. For more information, see the [JSON specification](#).

### Running your first query

You will begin by running basic queries with SELECT, WHERE, and FROM clauses.

### Open Data Explorer

1. In the **Azure Cosmos DB** blade, locate and select the **Data Explorer** link on the left side of the blade.
2. In the **Data Explorer** section, expand the **NutritionDatabase** database node and then expand the **FoodCollection** container node.
3. Within the **FoodCollection** node, select the **Items** link.
4. View the items within the container. Observe how these documents have many properties, including arrays.

The screenshot shows the SQL API interface. On the left, the 'NutritionDatabase' is expanded, showing 'FoodCollection' and 'Items'. The 'Items' table is selected, and the item with 'id' 23231, 'Beef Products', is highlighted. The right pane displays the JSON data for this item:

```

1 {
2   "id": "23231",
3   "description": "Beef, rib eye steak, boneless, lip off, separable",
4   "tags": [
5     {
6       "name": "beef"
7     },
8     {
9       "name": "rib eye steak"
10    },
11    {
12      "name": "boneless"
13    },
14    {
15      "name": "lip off"
16    },
17    {
18      "name": "separable lean and fat"
19    },
20    {
21      "name": "trimmed to 0\" fat"
22    },
23    {
24      "name": "choice"
25    },
26    {
27      "name": "raw"
28    }
29  ],
30   "unspec": "..."

```

5. Select **New SQL Query**.

The screenshot shows the SQL API interface with the 'Items' table selected. The 'New SQL Query' option in the context menu is highlighted with a red box. The menu options are:

- New SQL Query
- New Stored Procedure
- New UDF
- New Trigger
- Delete Container

6. Paste the following SQL query and select **Execute Query**.

```
SELECT *
FROM food
WHERE food.foodGroup = "Snacks" and food.id = "19015"
```

7. You will see that the query returned the single document where id is "19015" and the foodGroup is "Snacks". Explore the structure of this item as it is representative of the items within the **FoodCollection** container that we will be working with for the remainder of this section.

Items

Items

Query 1 ×

1

2

3

SELECT \*

FROM food

WHERE food.foodGroup = "Snacks" and food.id = "19015"

Results

Query Stats

1 - 1

[

{

"id": "19015",

"description": "Snacks, granola bars, hard, plain",

"tags": [

{

"name": "snacks"

},

{

"name": "granola bars"

},

{

"name": "hard"

},

{

"name": "plain"

}

],

"version": 1,

"isFromSurvey": false,

"foodGroup": "Snacks",

"nutrients": [

## Advanced projection

Azure Cosmos DB supports several forms of transformation on the resultant JSON. One of the simplest is to alias your JSON elements using the AS aliasing keyword as you project your results.

By running the query below you will see that the element names are transformed. In addition, the projection is accessing only the first element in the servings array for all items specified by the WHERE clause.

Run the below query by selecting the **New SQL Query**.

Paste the following SQL query and then select **Execute Query**.

```
SELECT food.description,  
food.foodGroup,  
food.servings[0].description AS servingDescription,  
food.servings[0].weightInGrams AS servingWeight  
FROM food  
WHERE food.foodGroup = "Fruits and Fruit Juices"  
AND food.servings[0].description = "cup"
```

The screenshot shows the Azure Cosmos DB query editor interface. At the top, there are tabs for 'FoodCollection ...' and 'Query 1'. The 'Query 1' tab is active, displaying a SQL query:

```
1 SELECT food.description,  
2 food.foodGroup,  
3 food.servings[0].description AS servingDescription,  
4 food.servings[0].weightInGrams AS servingWeight  
5 FROM food  
6 WHERE food.foodGroup = "Fruits and Fruit Juices"  
7 AND food.servings[0].description = "cup"  
8
```

Below the query editor, there are tabs for 'Results' and 'Query Stats'. The 'Results' tab is active, showing a list of 100 results. The first few results are displayed as JSON objects:

```
{  
  "description": "Pineapple juice, DOLE, canned, not from concentrate, unsweetened, with added vitamins A,  
  "foodGroup": "Fruits and Fruit Juices",  
  "servingDescription": "cup",  
  "servingWeight": 250  
},  
{  
  "description": "Apricot nectar, canned, without added ascorbic acid",  
  "foodGroup": "Fruits and Fruit Juices",  
  "servingDescription": "cup",  
  "servingWeight": 251  
},  
{  
  "description": "Pineapple juice, canned or bottled, unsweetened, without added ascorbic acid",  
  "foodGroup": "Fruits and Fruit Juices",  
  "servingDescription": "cup",  
  "servingWeight": 250  
},  
{  
  "description": "Apricots, dehydrated (low-moisture), sulfured, stewed",  
  "foodGroup": "Fruits and Fruit Juices",  
  "servingDescription": "cup",  
  "servingWeight": 249  
},  
{  
  "description": "Apricots, dehydrated (low-moisture), sulfured, uncooked",  
  "foodGroup": "Fruits and Fruit Juices",  
  "servingDescription": "cup",  
  "servingWeight": 119  
},  
{  
  "description": "Apple juice, canned or bottled, unsweetened, without added ascorbic acid",  
  "foodGroup": "Fruits and Fruit Juices",  
  "servingDescription": "cup",  
  "servingWeight": 248  
},  
{  
  "description": "Pear nectar, canned, with added ascorbic acid",  
  "foodGroup": "Fruits and Fruit Juices",  
  "servingDescription": "cup",  
  "servingWeight": 250  
},  
{  
  "description": "Pineapple juice, frozen concentrate, unsweetened, diluted with 3 volume water",  
  "foodGroup": "Fruits and Fruit Juices",  
  "servingDescription": "cup",  
  "servingWeight": 250  
}
```

## ORDER BY clause

Azure Cosmos DB supports adding an ORDER BY clause to sort results based on one or more properties

Run the below query by selecting the **New SQL Query**.

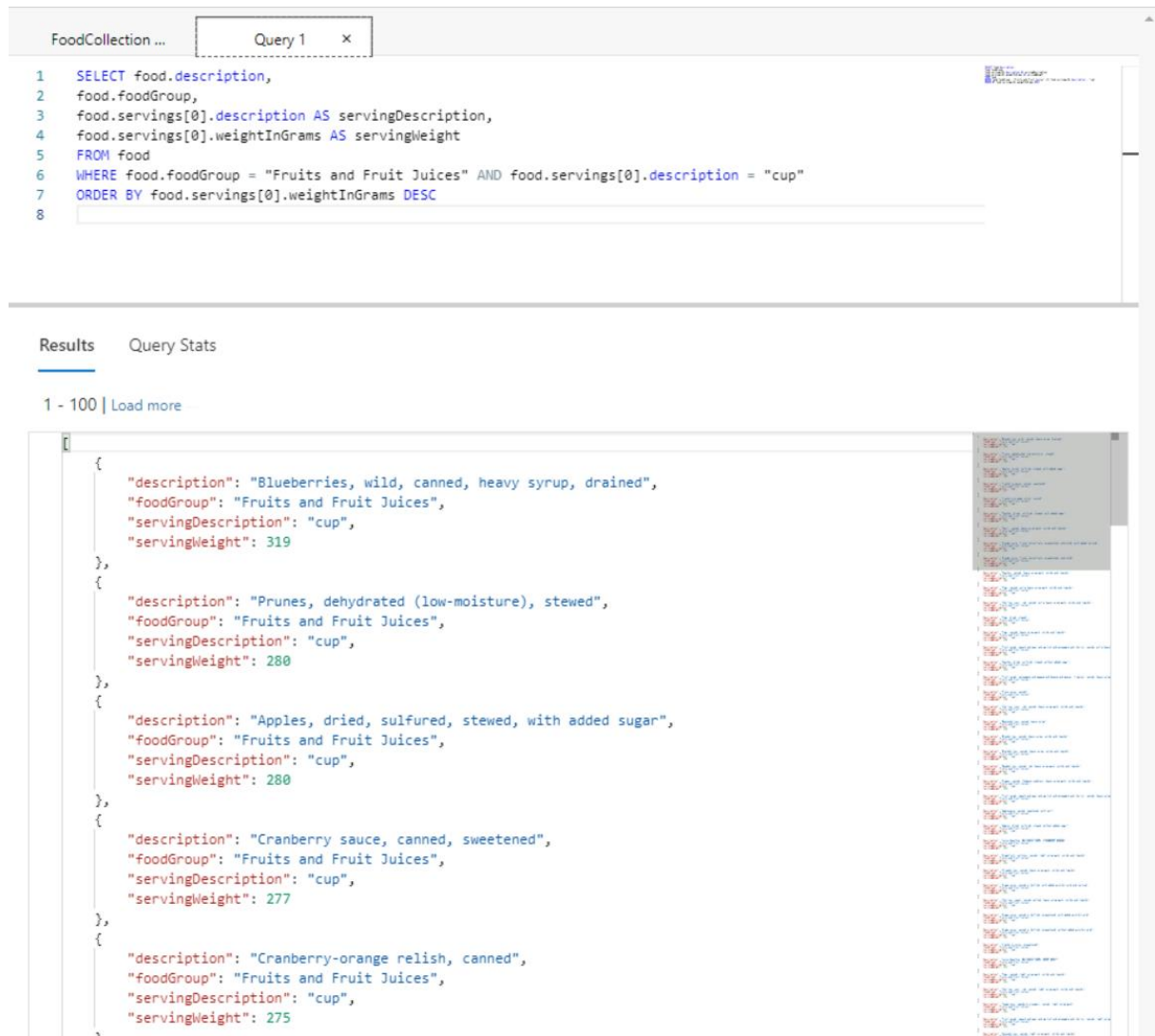
Paste the following SQL query and then select **Execute Query**.

```
SELECT food.description,  
food.foodGroup,  
food.servings[0].description AS servingDescription,  
food.servings[0].weightInGrams AS servingWeight
```

```

FROM food
WHERE food.foodGroup = "Fruits and Fruit Juices" AND
food.servings[0].description = "cup"
ORDER BY food.servings[0].weightInGrams DESC

```



The screenshot shows the Azure Cosmos DB Query Explorer interface. At the top, there's a tab for 'Query 1'. Below it, the SQL query is entered:

```

1 SELECT food.description,
2 food.foodGroup,
3 food.servings[0].description AS servingDescription,
4 food.servings[0].weightInGrams AS servingWeight
5 FROM food
6 WHERE food.foodGroup = "Fruits and Fruit Juices" AND food.servings[0].description = "cup"
7 ORDER BY food.servings[0].weightInGrams DESC
8

```

Below the query editor, there are tabs for 'Results' and 'Query Stats'. The 'Results' tab is active, showing a list of 100 results. The first few results are visible as JSON objects:

```

{
  "description": "Blueberries, wild, canned, heavy syrup, drained",
  "foodGroup": "Fruits and Fruit Juices",
  "servingDescription": "cup",
  "servingWeight": 319
},
{
  "description": "Prunes, dehydrated (low-moisture), stewed",
  "foodGroup": "Fruits and Fruit Juices",
  "servingDescription": "cup",
  "servingWeight": 280
},
{
  "description": "Apples, dried, sulfured, stewed, with added sugar",
  "foodGroup": "Fruits and Fruit Juices",
  "servingDescription": "cup",
  "servingWeight": 280
},
{
  "description": "Cranberry sauce, canned, sweetened",
  "foodGroup": "Fruits and Fruit Juices",
  "servingDescription": "cup",
  "servingWeight": 277
},
{
  "description": "Cranberry-orange relish, canned",
  "foodGroup": "Fruits and Fruit Juices",
  "servingDescription": "cup",
  "servingWeight": 275
}

```

You can learn more about configuring the required indexes for an Order By clause in the later Indexing Lab or by reading [our docs](#).

## Limiting query result size

Azure Cosmos DB supports the TOP keyword. TOP can be used to limit the number of returning values from a query.

Run the query below to see the top 20 results.

```

SELECT TOP 20 food.id,
food.description,
food.tags,
food.foodGroup
FROM food
WHERE food.foodGroup = "Snacks"

```

The screenshot shows a database interface with a query editor and a results pane. The query editor, titled 'Query 1', contains the following SQL code:

```
1 SELECT TOP 20 food.id,  
2 food.description,  
3 food.tags,  
4 food.foodGroup  
5 FROM food  
6 WHERE food.foodGroup = "Snacks"  
7
```

The results pane, titled 'Results', shows the first 20 results (1 - 20). The results are displayed as a JSON array of two objects, each representing a food item. The first object has an id of 42204, a description of 'Rice cake, cracker (include hain mini rice cakes)', and tags for 'rice cake' and 'cracker (include hain mini rice cakes)'. The second object has an id of 19800, a description of 'Snacks, corn cakes, very low sodium', and tags for 'snacks', 'corn cakes', and 'very low sodium'. Both items belong to the 'Snacks' food group.

The OFFSET LIMIT clause is an optional clause to skip then take some number of values from the query. The OFFSET count and the LIMIT count are required in the OFFSET LIMIT clause.

```
SELECT food.id,  
food.description,  
food.tags,  
food.foodGroup  
FROM food  
WHERE food.foodGroup = "Snacks"  
ORDER BY food.id  
OFFSET 10 LIMIT 10
```

The screenshot shows a database query interface. At the top, there's a tab labeled "Query 1" with a close button. Below the tab, a SQL query is entered in a text area:

```
1 SELECT food.id,  
2 food.description,  
3 food.tags,  
4 food.foodGroup  
5 FROM food  
6 WHERE food.foodGroup = "Snacks"  
7 ORDER BY food.id  
8 OFFSET 10 LIMIT 10  
9
```

Below the query editor, there are two tabs: "Results" and "Query Stats". The "Results" tab is active, showing "1 - 10" results. The results are displayed in a JSON format, showing two objects for food items with IDs 19015 and 19016. The first object has tags: "snacks", "granola bars", "hard", and "plain". The second object has tags: "snacks", "granola bars", "hard", and "almond". Both items belong to the "Snacks" food group.

When OFFSET LIMIT is used in conjunction with an ORDER BY clause, the result set is produced by doing skip and take on the ordered values. If no ORDER BY clause is used, it will result in a deterministic order of values.

## More advanced filtering

Let's add the IN and BETWEEN keywords into our queries. IN can be used to check whether a specified value matches any element in a given list and BETWEEN can be used to run queries against a range of values.

Run the query below:

```
SELECT food.id,  
food.description,
```



```

food.tags,
food.foodGroup,
food.version
FROM food
WHERE food.foodGroup IN ("Poultry Products", "Sausages and Luncheon Meats")
AND (food.id BETWEEN "05740" AND "07050")

```

The screenshot shows the Azure Cosmos DB Query Explorer interface. At the top, there's a tab for 'Query 1'. Below it, the SQL query is displayed in a text editor. The query selects food.id, food.description, food.tags, food.foodGroup, and food.version from the 'food' collection, filtered by foodGroup and id. Below the query editor, there are tabs for 'Results' and 'Query Stats'. The 'Results' tab is active, showing a list of 54 results. The first three results are displayed as JSON objects in a code editor. The first result is for food.id '07034', description 'Headcheese, pork', and foodGroup 'Sausages and Luncheon Meats'. The second result is for food.id '07032', description 'Ham and cheese loaf or roll', and foodGroup 'Sausages and Luncheon Meats'. The third result is for food.id '07028', description 'Ham, sliced, packaged (96% fat free, water added)', and foodGroup 'Sausages and Luncheon Meats'. The right side of the interface shows a tree view of the database structure.

```

{
  "id": "07034",
  "description": "Headcheese, pork",
  "tags": [
    {
      "name": "headcheese"
    },
    {
      "name": "pork"
    }
  ],
  "foodGroup": "Sausages and Luncheon Meats",
  "version": 1
},
{
  "id": "07032",
  "description": "Ham and cheese loaf or roll",
  "tags": [
    {
      "name": "ham and cheese loaf or roll"
    }
  ],
  "foodGroup": "Sausages and Luncheon Meats",
  "version": 1
},
{
  "id": "07028",
  "description": "Ham, sliced, packaged (96% fat free, water added)",
  "tags": [
    {
      "name": "ham"
    },
    {
      "name": "sliced"
    },
    {
      "name": "packaged (96% fat free"
    },
    {
      "name": "water added)"
    }
  ],
  "foodGroup": "Sausages and Luncheon Meats",
  "version": 1
}

```

## More advanced projection

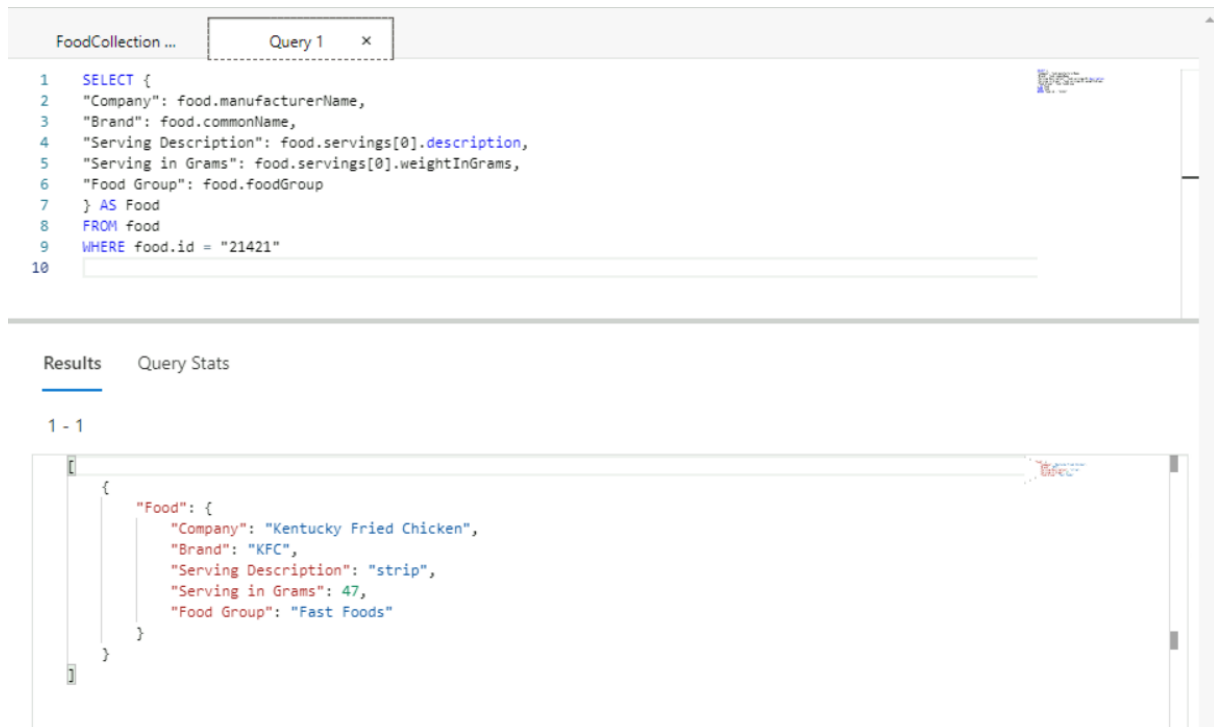
Azure Cosmos DB supports JSON projection within its queries. Let's project a new JSON Object with modified property names.

Run the query below to see the results.

```

SELECT {
  "Company": food.manufacturerName,
  "Brand": food.commonName,
  "Serving Description": food.servings[0].description,
  "Serving in Grams": food.servings[0].weightInGrams,
  "Food Group": food.foodGroup
} AS Food
FROM food
WHERE food.id = "21421"

```



## JOIN within your documents

Azure Cosmos DB's JOIN supports intra-document and self-joins. Azure Cosmos DB does not support JOINS across documents or containers.

In an earlier query example we returned a result with attributes of just the first serving of the `food.servings` array. By using the join syntax below, we can now return an item in the result for every item within the serving array while still being able to project the attributes from elsewhere in the item.

Run the query below to iterate on the food document's servings.

```

SELECT
  food.id as FoodID,
  serving.description as ServingDescription
FROM food
JOIN serving IN food.servings
WHERE food.id = "03226"

```

The screenshot shows the Azure Cosmos DB query editor interface. At the top, there are tabs for 'FoodCollection ...' and 'Query 1'. The query editor contains the following SQL code:

```
1 SELECT
2 food.id as FoodID,
3 serving.description as ServingDescription
4 FROM food
5 JOIN serving IN food.servings
6 WHERE food.id = "03226"
7
```

Below the query editor, there are tabs for 'Results' and 'Query Stats'. The 'Results' tab is selected, showing the results of the query. The results are displayed as a JSON array with two objects:

```
1 - 2
[
  {
    "FoodID": "03226",
    "ServingDescription": "oz"
  },
  {
    "FoodID": "03226",
    "ServingDescription": "jar"
  }
]
```

JOINS are useful if you need to filter on properties within an array. Run the below example that has filter after the intra-document JOIN.

```
SELECT VALUE COUNT(1)
FROM c
JOIN t IN c.tags
JOIN s IN c.servings
WHERE t.name = 'infant formula' AND s.amount > 1
```

The screenshot shows the Azure Cosmos DB query editor interface. At the top, there are tabs for 'FoodCollection ...' and 'Query 1'. The query editor contains the following SQL code:

```
1 SELECT VALUE COUNT(1)
2 FROM c
3 JOIN t IN c.tags
4 JOIN s IN c.servings
5 WHERE t.name = 'infant formula' AND s.amount > 1
6
```

Below the query editor, there are tabs for 'Results' and 'Query Stats'. The 'Results' tab is selected, showing the results of the query. The results are displayed as a single value:

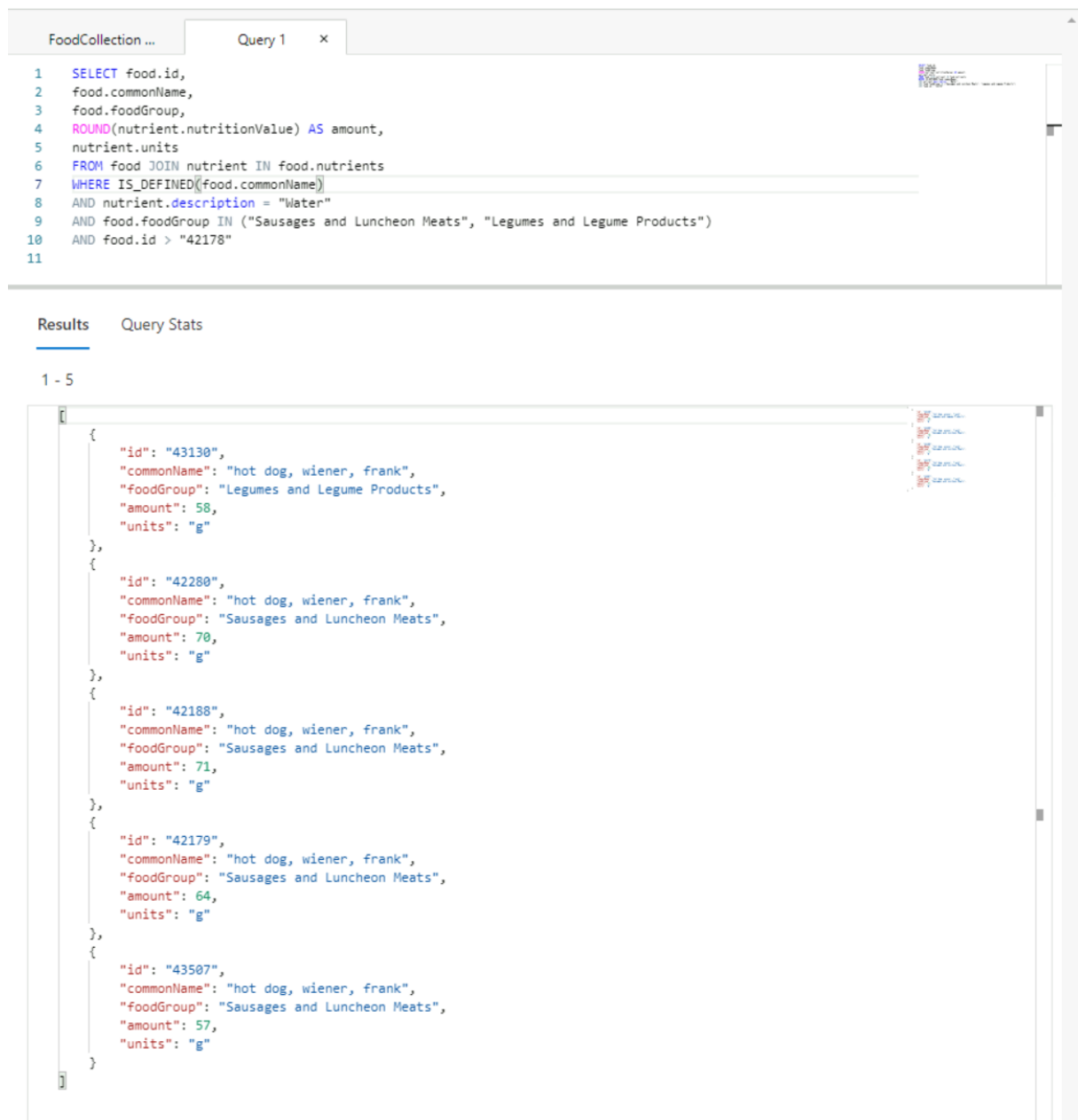
```
1 - 1
19
```

## System functions

Azure Cosmos DB supports a number of built-in functions for common operations. They cover mathematical functions like ABS, FLOOR and ROUND and type checking functions like IS\_ARRAY, IS\_BOOL and IS\_DEFINED. [Learn more about supported system functions.](#)

Run the query below to see example use of some system functions

```
SELECT food.id,  
food.commonName,  
food.foodGroup,  
ROUND(nutrient.nutritionValue) AS amount,  
nutrient.units  
FROM food JOIN nutrient IN food.nutrients  
WHERE IS_DEFINED(food.commonName)  
AND nutrient.description = "Water"  
AND food.foodGroup IN ("Sausages and Luncheon Meats", "Legumes and Legume  
Products")  
AND food.id > "42178"
```



The screenshot displays a database interface with two main panes. The top pane, titled 'Query 1', contains a SQL query. The bottom pane, titled 'Results', shows the output of the query as a JSON array of five objects, each representing a food item with its nutrients.

**Query:**

```
1 SELECT food.id,  
2 food.commonName,  
3 food.foodGroup,  
4 ROUND(nutrient.nutritionValue) AS amount,  
5 nutrient.units  
6 FROM food JOIN nutrient IN food.nutrients  
7 WHERE IS_DEFINED(food.commonName)  
8 AND nutrient.description = "Water"  
9 AND food.foodGroup IN ("Sausages and Luncheon Meats", "Legumes and Legume Products")  
10 AND food.id > "42178"  
11
```

**Results (1 - 5):**

```
{  
  "id": "43130",  
  "commonName": "hot dog, wiener, frank",  
  "foodGroup": "Legumes and Legume Products",  
  "amount": 58,  
  "units": "g"  
},  
{  
  "id": "42280",  
  "commonName": "hot dog, wiener, frank",  
  "foodGroup": "Sausages and Luncheon Meats",  
  "amount": 70,  
  "units": "g"  
},  
{  
  "id": "42188",  
  "commonName": "hot dog, wiener, frank",  
  "foodGroup": "Sausages and Luncheon Meats",  
  "amount": 71,  
  "units": "g"  
},  
{  
  "id": "42179",  
  "commonName": "hot dog, wiener, frank",  
  "foodGroup": "Sausages and Luncheon Meats",  
  "amount": 64,  
  "units": "g"  
},  
{  
  "id": "43507",  
  "commonName": "hot dog, wiener, frank",  
  "foodGroup": "Sausages and Luncheon Meats",  
  "amount": 57,  
  "units": "g"  
}
```

## Correlated sub-queries

In many scenarios, a sub-query may be effective. A correlated sub-query is a query that references values from an outer query. We will walk through some of the most useful examples here. You can [learn more about subqueries](#).

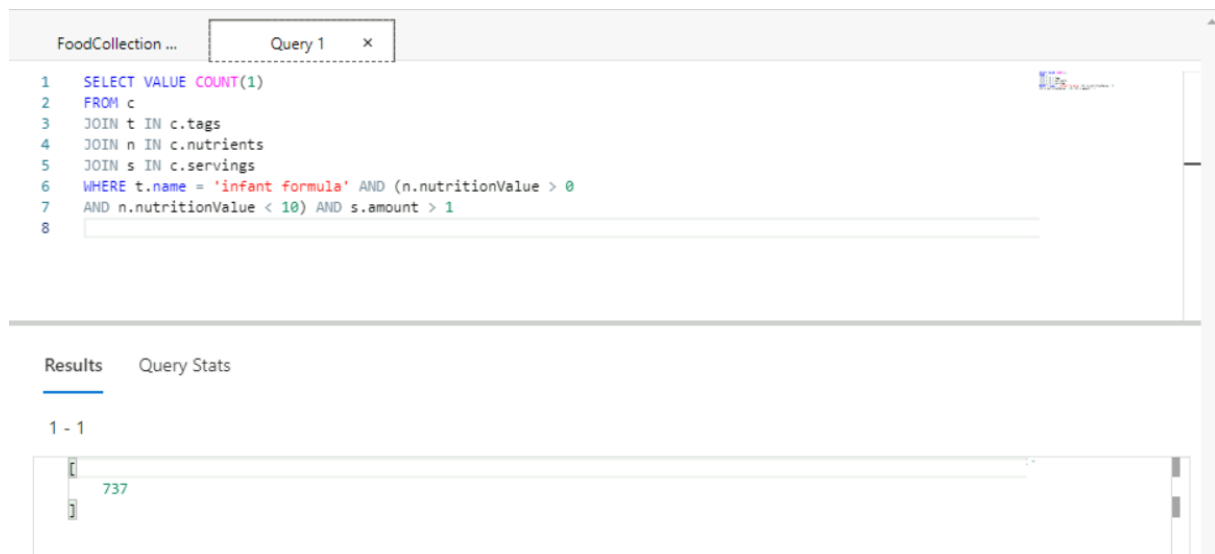
There are two types of sub-queries: Multi-value sub-queries and scalar sub-queries. Multi-value sub-queries return a set of documents and are always used within the FROM clause. A scalar sub-query expression is a sub-query that evaluates to a single value.

## Multi-value subqueries

You can optimize JOIN expressions with a sub-query.

Consider the following query which performs a self-join and then applies a filter on name, nutritionValue, and amount. We can use a sub-query to filter out the joined array items before joining with the next expression.

```
SELECT VALUE COUNT(1)
FROM c
JOIN t IN c.tags
JOIN n IN c.nutrients
JOIN s IN c.servings
WHERE t.name = 'infant formula' AND (n.nutritionValue > 0
AND n.nutritionValue < 10) AND s.amount > 1
```



We could rewrite this query using three sub-queries to optimize and reduce the Request Unit (RU) charge. Observe that the multi-value sub-query always appears in the FROM clause of the outer query.

```
SELECT VALUE COUNT(1)
FROM c
JOIN (SELECT VALUE t FROM t IN c.tags WHERE t.name = 'infant formula')
JOIN (SELECT VALUE n FROM n IN c.nutrients WHERE n.nutritionValue > 0 AND
n.nutritionValue < 10)
JOIN (SELECT VALUE s FROM s IN c.servings WHERE s.amount > 1)
```

The screenshot shows a database query editor with a tab labeled "Query 1". The query is as follows:

```
1 SELECT VALUE COUNT(1)
2 FROM c
3 JOIN (SELECT VALUE t FROM t IN c.tags WHERE t.name = 'infant formula')
4 JOIN (SELECT VALUE n FROM n IN c.nutrients WHERE n.nutritionValue > 0 AND n.nutritionValue < 10)
5 JOIN (SELECT VALUE s FROM s IN c.servings WHERE s.amount > 1)
6
```

Below the query editor, there are two tabs: "Results" and "Query Stats". The "Results" tab is active, showing "1 - 1" and a single result value of 737.

## Scalar sub-queries

One use case of scalar sub-queries is rewriting `ARRAY_CONTAINS` as `EXISTS`.

Consider the following query that uses `ARRAY_CONTAINS`:

```
SELECT TOP 5 f.id, f.tags
FROM food f
WHERE ARRAY_CONTAINS(f.tags, {name: 'orange'})
```

FoodCollection ... Query 1 x

```
1 SELECT TOP 5 f.id, f.tags
2 FROM food f
3 WHERE ARRAY_CONTAINS(f.tags, {name: 'orange'})
4
```

Results Query Stats

1 - 5

```
{
  "id": "15232",
  "tags": [
    {
      "name": "fish"
    },
    {
      "name": "roughy"
    },
    {
      "name": "orange"
    },
    {
      "name": "cooked"
    },
    {
      "name": "dry heat"
    }
  ]
},
{
  "id": "19873",
  "tags": [
    {
      "name": "frozen novelties"
    },
    {
      "name": "ice type"
    },
    {
      "name": "sugar free"
    },
    {
      "name": "orange"
    },
    {
      "name": "cherry"
    },
    {
      "name": "and grape popsicle pops"
    }
  ]
}
```

Run the following query which has the same results but uses EXISTS:

```
SELECT TOP 5 f.id, f.tags
FROM food f
WHERE EXISTS(SELECT VALUE t FROM t IN f.tags WHERE t.name = 'orange')
```

The screenshot shows a database query editor with a tab labeled "Query 1". The SQL query is as follows:

```
1 SELECT TOP 5 f.id, f.tags
2 FROM food f
3 WHERE EXISTS(SELECT VALUE t FROM t IN f.tags WHERE t.name = 'orange')
4
```

Below the query editor, the "Results" tab is active, displaying the first five results of the query. The results are shown in a JSON format:

```
1 - 5
[
  {
    "id": "15232",
    "tags": [
      { "name": "fish" },
      { "name": "roughy" },
      { "name": "orange" },
      { "name": "cooked" },
      { "name": "dry heat" }
    ]
  },
  {
    "id": "19873",
    "tags": [
      { "name": "frozen novelties" },
      { "name": "ice type" }
    ]
  }
]
```

On the right side of the results, there is a vertical pane showing a tree view of the JSON structure, with nodes for "id" and "tags" expanded.

The major advantage of using EXISTS is the ability to have complex filters in the EXISTS function, rather than just the simple equality filters which ARRAY\_CONTAINS permits.

Here is an example:

```
SELECT VALUE c.description
FROM c
JOIN n IN c.nutrients
WHERE n.units= "mg" AND n.nutritionValue > 0
```





properties from the index, modification is possible through the Azure Portal, ARM template, PowerShell, Azure CLI or any Cosmos DB SDK.

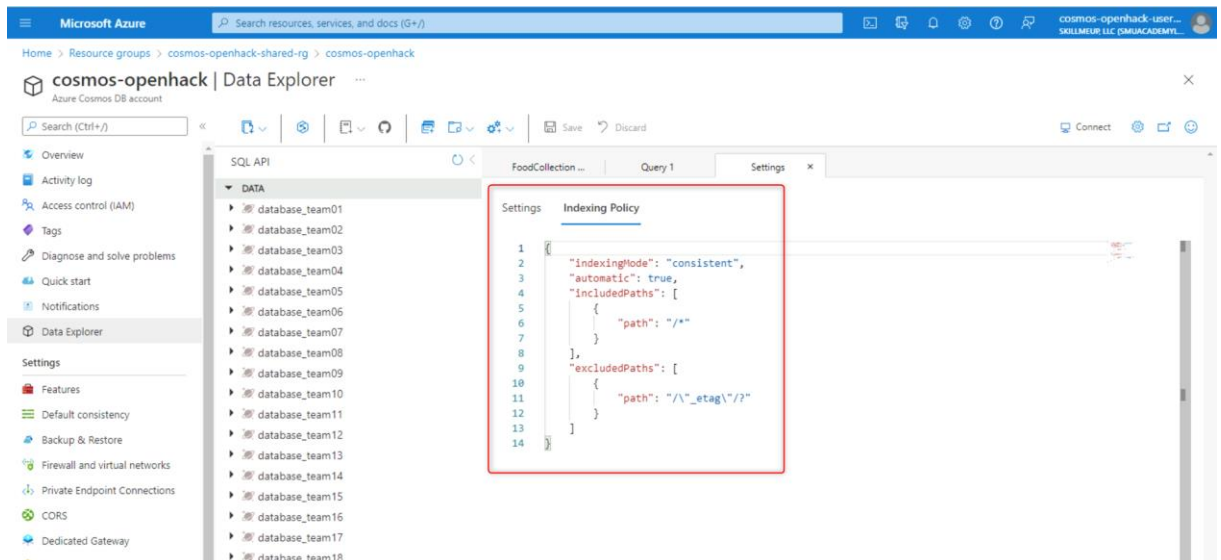
Azure Cosmos DB uses an inverted index, representing your data in a tree form. For a brief introduction on how this works, read our [indexing overview](#) before continuing with the lab.

## Customizing the indexing policy

In this lab section, you will view and modify the indexing policy for your **FoodCollection**.

### Open Data Explorer

1. On the left side of the portal, select the **Resource groups** link.
2. In the **Resource groups** blade, locate and select the **cosmos-openhack-rg** resource group.
3. In the **cosmos-openhack** blade, select your **Azure Cosmos DB** account.
4. In the **Azure Cosmos DB** blade, locate and select the **Data Explorer** link on the left side of the blade.
5. In the **Data Explorer** section, expand the database **database\_teamxx** node and then expand the **FoodCollection** container node.
6. Within the **FoodCollection** node, select the **Items** link.
7. View the items within the container. Observe how these documents have many properties, including arrays. If we do not use a particular property in the WHERE clause, ORDER BY clause, or a JOIN, indexing the property does not provide any performance benefit.
8. Still within the **FoodCollection** node, select the **Settings** link.
9. Review the **Indexing Policy** section
  - Notice you can edit the JSON file that defines your container's index
  - An Indexing policy can also be modified through any Azure Cosmos DB SDK as well as ARM template, PowerShell or Azure CLI
  - During this lab we will modify the indexing policy through the Azure Portal



## Including and excluding Indexes

Instead of including an index on every property by default, you can choose to either include or exclude specific paths from the index. Let's go through some simple examples (no need to enter these into the Azure Portal, we can just review them here).

Within the **FoodCollection**, documents have this schema (some properties were removed for simplicity):

```
{
  "id": "36000",
  "_rid": "LYwNAKzLG9ADAAAAAAAAA==",
  "_self":
  "dbs/LYwNAA==/colls/LYwNAKzLG9A=/docs/LYwNAKzLG9ADAAAAAAAAA==/",
  "_etag": "\"0b008d85-0000-0700-0000-5d1a47e60000\"",
  "description": "APPLEBEE'S, 9 oz house sirloin steak",
  "tags": [
    {
      "name": "applebee's"
    },
    {
      "name": "9 oz house sirloin steak"
    }
  ],
  "manufacturerName": "Applebee's",
  "foodGroup": "Restaurant Foods",
  "nutrients": [
    {
      "id": "301",
      "description": "Calcium, Ca",
      "nutritionValue": 16,
      "units": "mg"
    },
    {
      "id": "312",
      "description": "Copper, Cu",
      "nutritionValue": 0.076,
      "units": "mg"
    }
  ]
}
```

```

    },
  ],
}

```

If you wanted to only index the `manufacturerName`, `foodGroup`, and `nutrients` array, you should define the following index policy. In this example, we use the wildcard character `*` to indicate that we would like to index all paths within the `nutrients` array.

```

{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {
      "path": "/manufacturerName/*"
    },
    {
      "path": "/foodGroup/*"
    },
    {
      "path": "/nutrients/[]/*"
    }
  ],
  "excludedPaths": [
    {
      "path": "/*"
    }
  ]
}

```

However, it's possible we may just want to index the `nutritionValue` of each array element.

In this next example, the indexing policy would explicitly specify that the `nutritionValue` path in the `nutrition` array should be indexed. Since we don't use the wildcard character `*`, no additional paths in the array are indexed.

```

{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {
      "path": "/manufacturerName/*"
    },
    {
      "path": "/foodGroup/*"
    },
    {
      "path": "/nutrients/[]/nutritionValue/*"
    }
  ],
  "excludedPaths": [
    {
      "path": "/*"
    }
  ]
}

```

Finally, it's important to understand the difference between the `*` and `?` characters. The `*` character indicates that Azure Cosmos DB should index every path beyond that specific node.

The ? character indicates that Azure Cosmos DB should index no further paths beyond this node.

In the above example, there are no additional paths under nutritionValue. If we were to modify the document and add a path here, having the wildcard character \* in the above example would ensure that the property is indexed without explicitly mentioning the name.

## Understand query requirements

Before modifying indexing policy, it's important to understand how the data is used in the collection.

If your workload is write-heavy or your documents are large, you should only index necessary paths. This will significantly decrease the amount of RU's required for inserts, updates, and deletes.

Let's imagine that the following queries are the only read operations that are executed on the **FoodCollection** container.

### Query #1

```
SELECT * FROM c WHERE c.manufacturerName = <manufacturerName>
```

### Query #2

```
SELECT * FROM c WHERE c.foodGroup = <foodGroup>
```

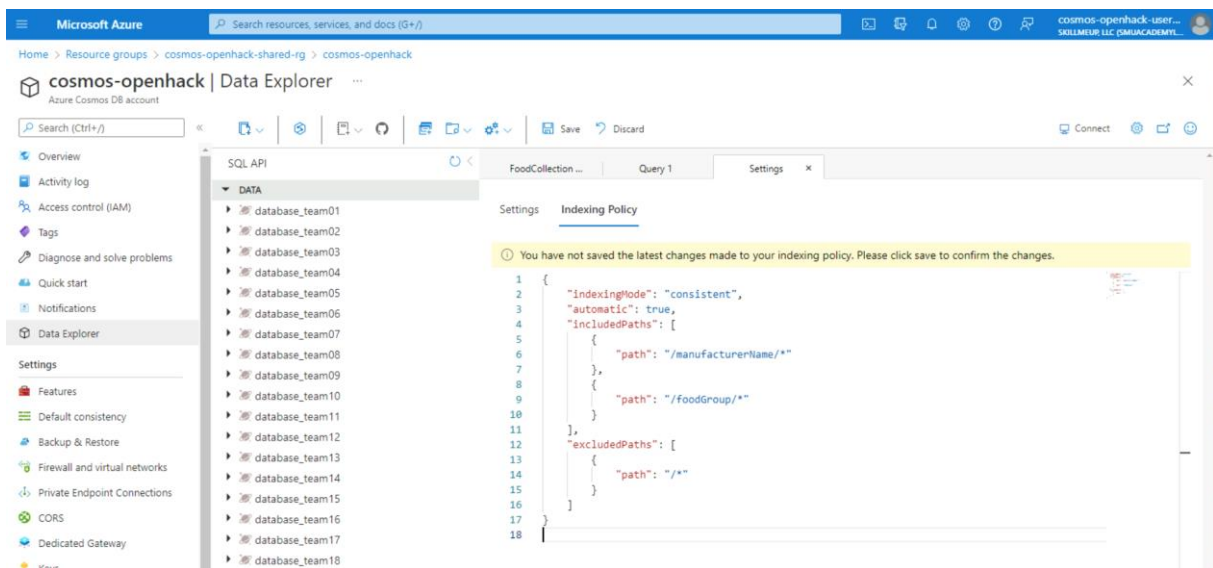
These queries only require an index be defined on **manufacturerName** and **foodGroup**. We can modify the indexing policy to index only these properties.

## Edit the indexing policy by including paths

1. In the Azure Portal, navigate back to the **FoodCollection** container
2. Select the **Settings** link
3. In the **Indexing Policy** section, replace the existing json file with the following:

```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {
      "path": "/manufacturerName/*"
    },
    {
      "path": "/foodGroup/*"
    }
  ],
  "excludedPaths": [
    {
      "path": "/*"
    }
  ]
}
```

}



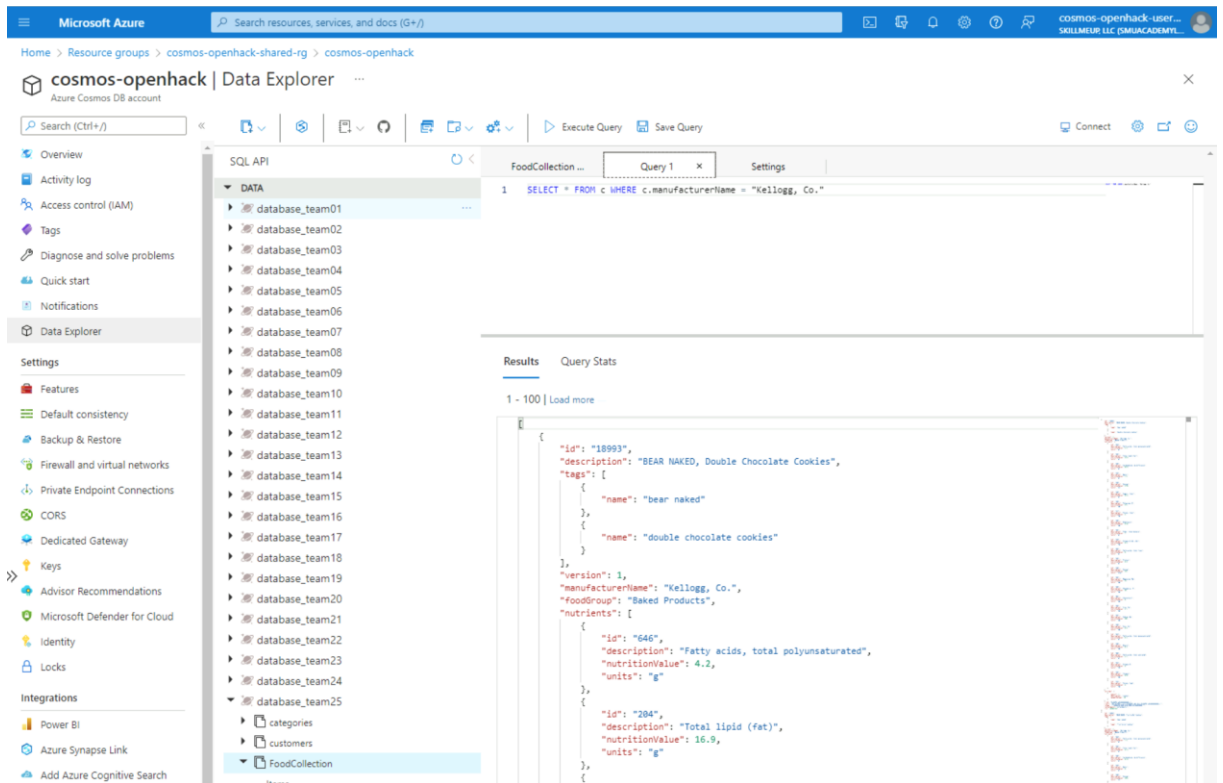
This new indexing policy will create an index on only the manufacturerName and foodGroup properties. It will remove indexes on all other properties.

4. Select **Save**. Azure Cosmos DB will update the index in the container, using your excess provisioned throughput to make the updates.

During the container re-indexing, write performance is unaffected. Queries run during the index update will not use the new index policy until it has been rebuilt.

5. In the menu, select the **New SQL Query** icon.
6. Paste the following SQL query and select **Execute Query**:

```
SELECT * FROM c WHERE c.manufacturerName = "Kellogg, Co."
```



7. Navigate to the **Query Stats** tab. You should observe that this query still has a low RU charge, even after removing some properties from the index. Because the **manufacturerName** was the only property used as a filter in the query, it was the only index that was required.

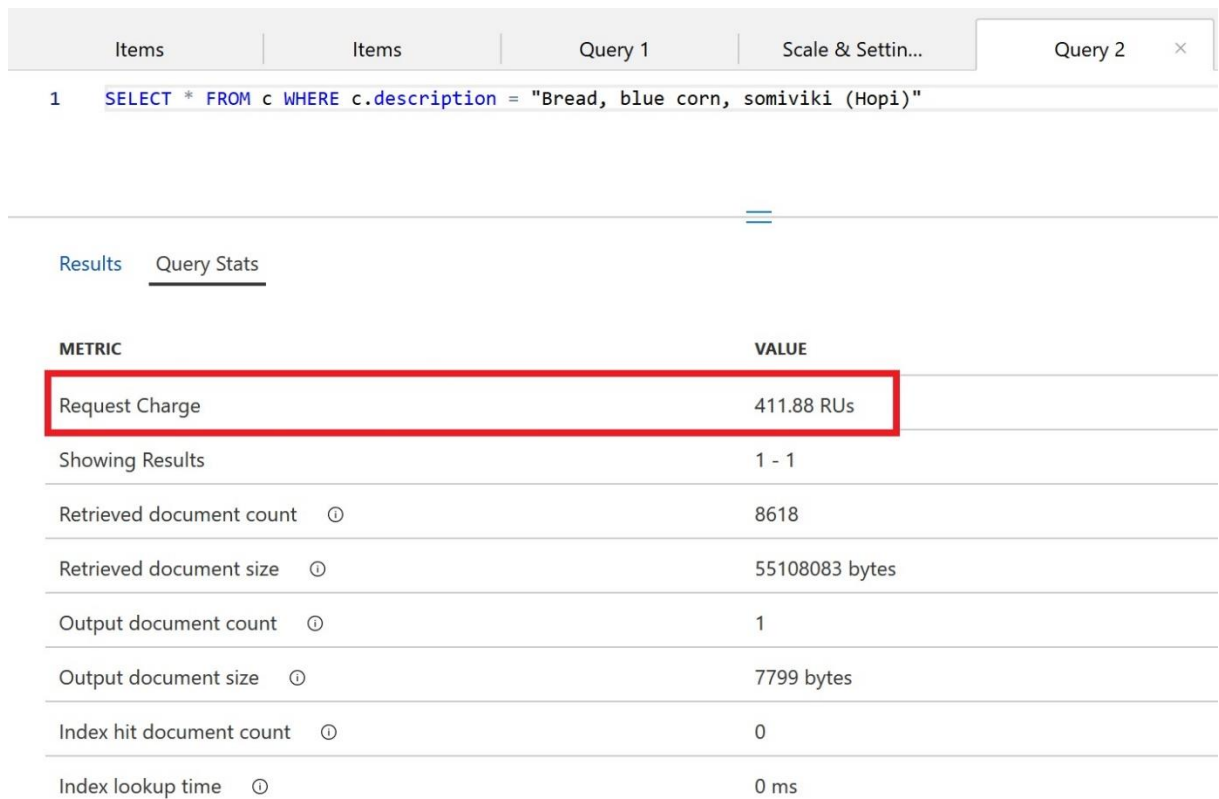
Items	Items	Query 1	Scale & Settin...	Query 2
1		SELECT * FROM c WHERE c.manufacturerName = "Kellogg, Co."		
Results	Query Stats			
METRIC	VALUE			
Request Charge	34.269999999999996 RUs			
Showing Results	1 - 100			
Retrieved document count ⓘ	200			
Retrieved document size ⓘ	689425 bytes			
Output document count ⓘ	200			
Output document size ⓘ	689725 bytes			
Index hit document count ⓘ	200			

8. Replace the query text with the following and select **Execute Query**:

```
SELECT * FROM c WHERE c.description = "Bread, blue corn, somiviki (Hopi)"
```

9. Observe that this query has a very high RU charge even though only a single document is returned. This is because no index is currently defined for the description property.

10. Observe the **Query Metrics**:



METRIC	VALUE
Request Charge	411.88 RUs
Showing Results	1 - 1
Retrieved document count ⓘ	8618
Retrieved document size ⓘ	55108083 bytes
Output document count ⓘ	1
Output document size ⓘ	7799 bytes
Index hit document count ⓘ	0
Index lookup time ⓘ	0 ms

If a query does not use the index, the **Index hit document count** will be 0. We can see above that the query needed to retrieve 8,618 documents and ultimately ended up only returning 1 document.

## Edit the indexing policy by excluding paths

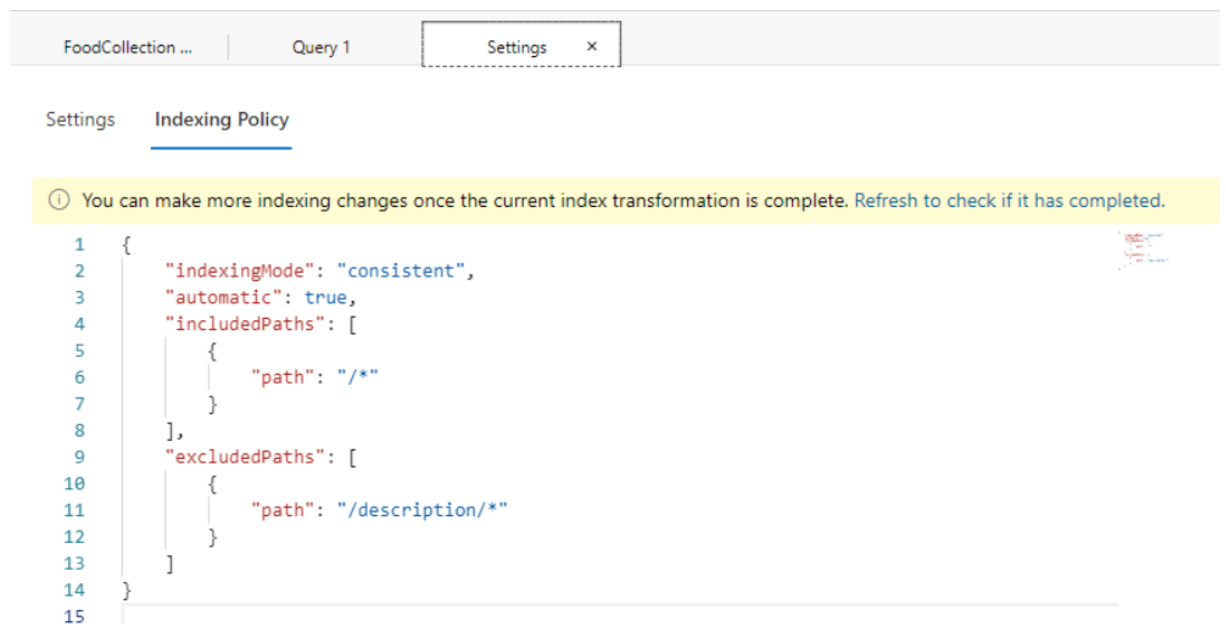
In addition to manually including certain paths to be indexed, you can exclude specific paths. In many cases, this approach can be simpler since it will allow all new properties in your document to be indexed by default. If there is a property that you are certain you will never use in your queries, you should explicitly exclude this path.

We will create an indexing policy to index every path except for the **description** property.

1. Navigate back to the **FoodCollection** in the Azure Portal
2. Select the **Settings** link
3. In the **Indexing Policy** section, replace the existing json file with the following:



```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {
      "path": "/*"
    }
  ],
  "excludedPaths": [
    {
      "path": "/description/*"
    }
  ]
}
```



This new indexing policy will create an index on every property **except** for the description.

4. Select **Save**. Azure Cosmos DB will update the index in the container, using your excess provisioned throughput to make the updates.

During the container re-indexing, write performance is unaffected. Queries run during the index update will not use the new index policy until it has been rebuilt.

5. After defining the new indexing policy, navigate to your **FoodCollection** and select the **Add New SQL Query** icon. Paste the following SQL query and select **Execute Query**:

```
SELECT * FROM c WHERE c.manufacturerName = "Kellogg, Co."
```

6. Navigate to the **Query Stats** tab. You should observe that this query still has a low RU charge since manufacturerName is indexed.
7. Replace the query text with the following and select **Execute Query**:

```
SELECT * FROM c WHERE c.description = "Bread, blue corn, somiviki (Hopi)"
```

8. Observe that this query has a very high RU charge even though only a single document is returned. This is because the `description` property is explicitly excluded in the indexing policy.

## Adding a Composite Index

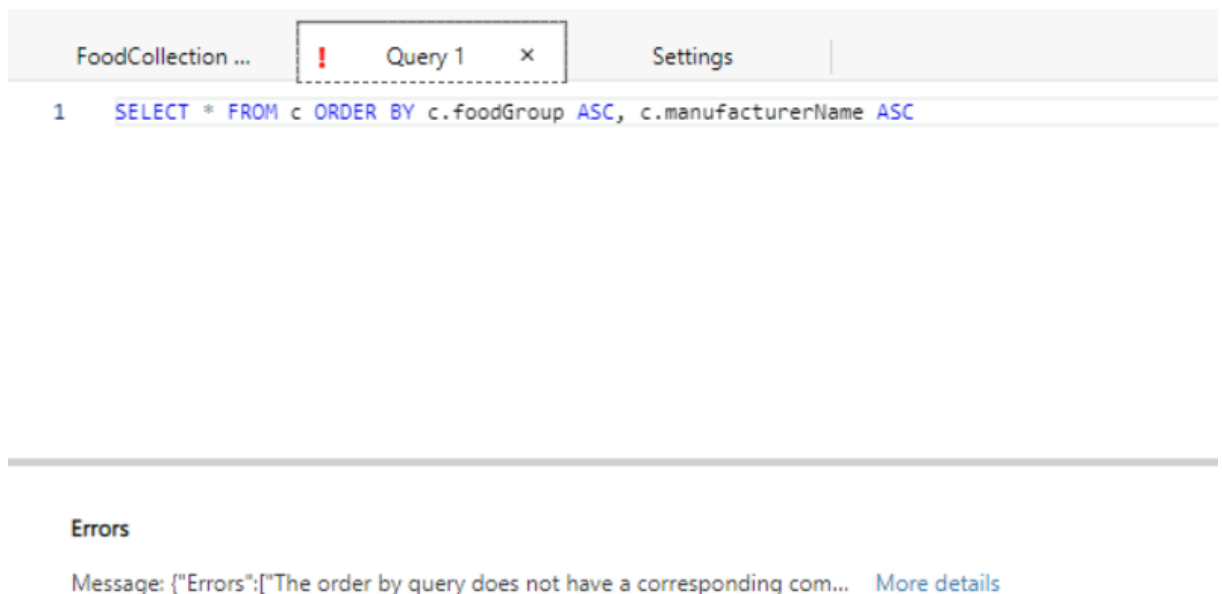
For ORDER BY queries that order by multiple properties, a composite index is required. A composite index is defined on multiple properties and must be manually created.

1. In the **Azure Cosmos DB** blade, locate and select the **Data Explorer** link on the left side of the blade.
2. In the **Data Explorer** section, expand the `database_teamxx` database node and then expand the **FoodCollection** container node.
3. Select the icon to add a **New SQL Query**.
4. Paste the following SQL query and select **Execute Query**

```
SELECT * FROM c ORDER BY c.foodGroup ASC, c.manufacturerName ASC
```

This query will fail with the following error:

"The order by query does not have a corresponding composite index that it can be served from."



In order to run a query that has an ORDER BY clause with one property, the default index is sufficient. Queries with multiple properties in the ORDER BY clause require a composite index.

5. Still within the **FoodCollection** node, select the **Settings** link. In the **Indexing Policy** section, you will add a composite index.
6. Replace the **Indexing Policy** with the following text:

```
{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {
      "path": "/manufacturerName/*"
    },
    {
      "path": "/foodGroup/*"
    }
  ],
  "excludedPaths": [
    {
      "path": "/*"
    },
    {
      "path": "/\"_etag\"/?"
    }
  ],
  "compositeIndexes": [
    [
      {
        "path": "/foodGroup",
        "order": "ascending"
      },
      {
        "path": "/manufacturerName",
        "order": "ascending"
      }
    ]
  ]
}
```

FoodCollection ... | Query 1 | Settings x

Settings | Indexing Policy

ⓘ You have not saved the latest changes made to your indexing policy. Please click save to confirm the changes.

```

1  {
2    "indexingMode": "consistent",
3    "automatic": true,
4    "includedPaths": [
5      {
6        "path": "/manufacturerName/*"
7      },
8      {
9        "path": "/foodGroup/*"
10     }
11   ],
12   "excludedPaths": [
13     {
14       "path": "/*"
15     },
16     {
17       "path": "/\"_etag\"/?"
18     }
19   ],
20   "compositeIndexes": [
21     [
22       {
23         "path": "/foodGroup",
24         "order": "ascending"
25       },
26       {
27         "path": "/manufacturerName",
28         "order": "ascending"
29       }
30     ]
31   ]
32 }
33

```

7. **Save** this new indexing policy. The update should take approximately 10-15 seconds to apply to your container.

This indexing policy defines a composite index that allows for the following ORDER BY queries. Test each of these by running them in your existing open query tab in the **Data Explorer**. When you define the order for properties in a composite index, they must either exactly match the order in the ORDER BY clause or be, in all cases, the opposite value.

8. Run the following query, which the current composite index does not support.

```
SELECT * FROM c ORDER BY c.foodGroup DESC, c.manufacturerName ASC
```

9. This query will not run without an additional composite index. Modify the indexing policy to include an additional composite index.

```

{
  "indexingMode": "consistent",
  "automatic": true,
  "includedPaths": [
    {

```

```

        "path": "/manufacturerName/*"
      },
      {
        "path": "/foodGroup/*"
      }
    ],
    "excludedPaths": [
      {
        "path": "/*"
      },
      {
        "path": "/\"_etag\"/?"
      }
    ],
    "compositeIndexes": [
      [
        {
          "path": "/foodGroup",
          "order": "ascending"
        },
        {
          "path": "/manufacturerName",
          "order": "ascending"
        }
      ],
      [
        {
          "path": "/foodGroup",
          "order": "descending"
        },
        {
          "path": "/manufacturerName",
          "order": "ascending"
        }
      ]
    ]
  }
}

```

10. Re-run the query, it should succeed.

FoodCollection ...

Query 1 ×

Settings

1 SELECT \* FROM c ORDER BY c.foodGroup DESC, c.manufacturerName ASC

Results

Query Stats

1 - 100 | Load more

```
{
  "id": "11873",
  "description": "Swamp cabbage, cooked, boiled, drained, with salt",
  "tags": [
    {
      "name": "swamp cabbage"
    },
    {
      "name": "cooked"
    },
    {
      "name": "boiled"
    },
    {
      "name": "drained"
    },
    {
      "name": "with salt"
    }
  ],
  "version": 1,
  "foodGroup": "Vegetables and Vegetable Products",
  "nutrients": [
    {
      "id": "307",
      "description": "Sodium, Na",
      "nutritionValue": 358,
      "units": "mg"
    },
    {
      "id": "309",
      "description": "Zinc, Zn",
      "nutritionValue": 0.16,
      "units": "mg"
    },
    {
      "id": "318",
```

[Learn more about defining composite indexes.](#)