Microsoft

Optimize query and operation performance in Azure Cosmos DB for NoSQL

Agenda

- Optimize indexes in Azure Cosmos DB for NoSQL
- Measure index performance in Azure Cosmos DB for NoSQL
- Implement integrated cache

Optimize indexes in Azure Cosmos DB for NoSQL

Index usage

The query engine will automatically try to use the most efficient method of evaluating filters, *index seek, index scan, full scan.*

Suppose the items in the *product* container are:

```
[
  { "id": "1", "name": "Mountain-400-W Silver", "price": 675.55 },
  { "id": "2", "name": "Touring-1000 Blue", "price": 1215.40 },
  { "id": "3", "name": "Road-200 Red", "price": 405.85 }
]
```

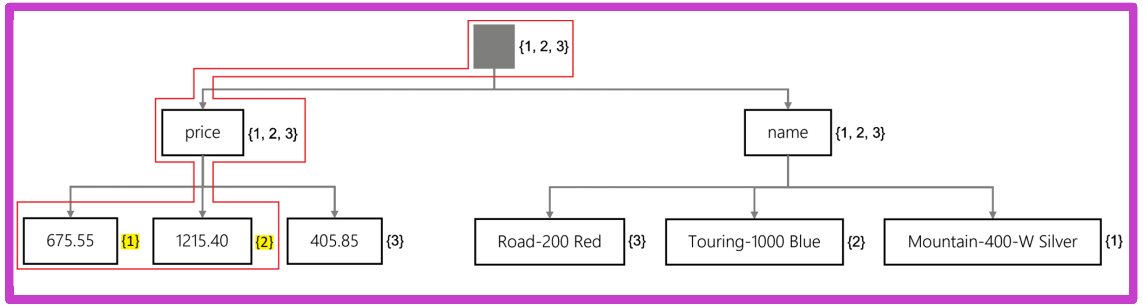
How will the index be used with the following queries?

```
SELECT *
FROM products p
WHERE p.name = 'Touring-1000 Blue'
```

```
SELECT *
FROM products p
WHERE p.name IN ('Road-200 Red', 'Mountain-400-W Silver')
```

```
SELECT *
FROM products p
WHERE p.price >= 500 AND p.price <= 1500
```

***product* inverted index tree:**



Review read-heavy index patterns

Read-centric workloads benefit from having an inverted index that includes as many fields as possible to maximize query performance and minimize request unit charges.

Consider this sample item in the *product* container. Consider that the applications querying items on this container never search or filter on the description or metadata properties.

```
{
    "id": "3324789",
    "name": "Road-200 Green",
    "price": 510.55,
    "description": "Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Cras faucibus, turpis ut pulvinar bibendum, sapien mauris fermentum magna, a
tincidunt magna diam tincidunt enim. Fusce convallis justo nulla, at
tristique diam tempus vel. Suspendisse potenti. Curabitur rhoncus neque vel
elit condimentum finibus. Nullam porta lorem vitae enim tincidunt elementum.
Vestibulum id felis sit amet neque commodo scelerisque. Suspendisse euismod
ex ut hendrerit eleifend. Quisque euismod consectetur vulputate.",
    "metadata": { "created_by": "sdfuouu",
                  "created_on": "2020-05-05T19:21:27.0000000Z",
                  "department": "cycling",
                  "sku": "RD200-G"
                }
}
```

Default index policy

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

Proposed index policies

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

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

Review write-heavy index patterns

Insert or update operations also make the indexer update the inverted index with data from your newly created or updated item. The more properties you index the more RUs used by the indexer.

Consider this sample item in the *product* container.

```
{
  "id": "3324734",
  "name": "Road-200 Green",
  "internal": {
     "tracking": { "id": "eac06d51-2462-4bfb-8eb6-46281da16f8e" } },
  "inStock": true,
  "price": 1303.33,
  "description": "Consequat dolore commodo tempor pariatur consectetur
fugiat labore velit aliqua ut anim. Et anim eu ea reprehenderit sit ullamco
elit irure laborum sunt ea adipisicing eu qui. Officia commodo ad amet ea
consectetur ea est fugiat.",
  "warehouse": { "shelfLocations": [ 20, 37, 35, 27, 38 ] },
  "metadata": { "color": "brown",
                "manufacturer": "Fabrikam",
                "supportEmail": "support@fabrik.am",
                "created_by": "sdfuouu",
                "created_on": "2020-05-05T19:21:27.0000000Z",
                "department": "cycling",
                "sku": "RD200-B" },
  "tags": [ "pariatur", "et", "commodo", "ex", "tempor", "esse",
            "nisi", "ullamco", "Lorem", "ullamco", "ex", "ea",
            "laborum", "tempor", "consequat" ]
}
```

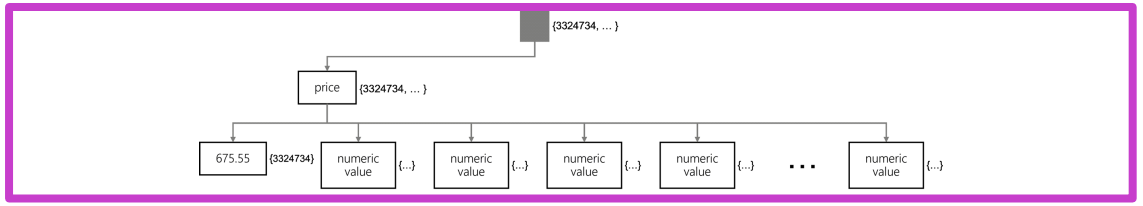
Assume the application only uses these two queries

```
SELECT *
FROM products p
WHERE p.price >= <numeric-value> AND p.price <= <numeric-value>
```

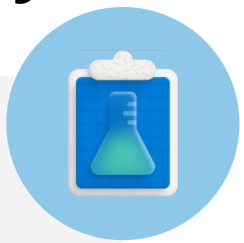
```
SELECT *
FROM products p
WHERE p.price = <numeric-value>
```

Proposed index policy

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



Lab – Optimize an Azure Cosmos DB for NoSQL container's index policy for common operations



- Prepare your development environment
- Create an Azure Cosmos DB for NoSQL account
- Run the test .NET application using the default indexing policy
- Update the indexing policy and rerun the .NET application

Measure index performance in Azure Cosmos DB for NoSQL

Enable indexing metrics

Azure Cosmos DB for NoSQL includes opt-in indexing metrics that illuminate how the current state of the index affects your query filters.

```
Container container = client.GetContainer("cosmicworks", "products");

string sql = "SELECT * FROM products p";

QueryDefinition query = new(sql);

//  PopulateIndexMetrics is disabled by default, enable it if troubleshooting query performance or are unsure how to modify your indexing policy.
QueryRequestOptions options = new()
{
   PopulateIndexMetrics = true
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOptions: options);

while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {
        Console.WriteLine($"[{product.id}]\t{product.name,35}\t{product.price,15:C}");
    }

    //  Do something with the metrics, in this example, we are sending it to the console output.
    Console.WriteLine(response.IndexMetrics);
}
```

Analyze indexing metrics results

Assume we are using the default index policy for the following queries.

Query 1

```
SELECT *
FROM products p
WHERE p.price > 500
```

Indexing metrics

```
Index Utilization Information
  Utilized Single Indexes
    Index Spec: /price/?
    Index Impact Score: High
    ---
  Potential Single Indexes
  Utilized Composite Indexes
  Potential Composite Indexes
```

Query 2

```
SELECT *
FROM products p
WHERE p.price > 500
    AND startsWith(p.name, 'Touring')
```

```
Index Utilization Information
  Utilized Single Indexes
    Index Spec: /price/?
    Index Impact Score: High
    ---
    Index Spec: /name/?
    Index Impact Score: High
    ---
  Potential Single Indexes
  Utilized Composite Indexes
  Potential Composite Indexes
```

Analyze indexing metrics results – Composite indexes

The indexing metrics could recommend we create a composite index.

Query 3

```
SELECT *
FROM products p
WHERE p.price > 500
    AND p.categoryName = 'Bikes, Touring Bikes'
```

Indexing metrics

```
Index Utilization Information
  Utilized Single Indexes
    Index Spec: /price/?
    Index Impact Score: High
    ---
    Index Spec: /categoryName/?
    Index Impact Score: High
    ---
  Potential Single Indexes
  Utilized Composite Indexes
  Potential Composite Indexes
    Index Spec: /categoryName ASC, /price ASC
    Index Impact Score: High
    ---
```

Add the potential composite index and run the query again.

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

```
Index Utilization Information
  Utilized Single Indexes
  Potential Single Indexes
  Utilized Composite Indexes
    Index Spec: /categoryName ASC, /price ASC
    Index Impact Score: High
    ---
  Potential Composite Indexes
```

Measure query cost

The QueryRequestOptions class is also helpful in measuring the cost of a query in RU/s.

```
Container container = client.GetContainer("cosmicworks", "products");
 string sql = "SELECT * FROM products p";
QueryDefinition query = new(sql);

// Set the MaxItemCount property of the QueryRequestOptions class to the number // of items you would like to return in each result page.
QueryRequestOptions options = new()
{
   MaxItemCount = 25
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOptions: options);

double totalRUs = 0;
while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {  // Do something with each product
    }
    //  Outputs the RU/s cost for returning every 25-item iteration.
    Console.WriteLine($"RU/s:\t\t{response.RequestCharge:0.00}");
    totalRUs += response.RequestCharge;
}

//  Returns the total RU/s cost of returning all items in the container..
Console.WriteLine($"Total RUs:\t{totalRUs:0.00}");
```

Sample console output

```
RUs: 2.82
RUs: 2.82
RUs: 2.83
RUs: 2.84
RUs: 2.25
Total RUs: 13.56
```

Measure point operation cost

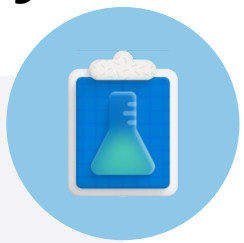
You can also use the .NET SDK to measure the cost, in RU/s, of individual operations.

```
Container container = client.GetContainer("cosmicworks", "products");

Product item = new(
    $"{Guid.NewGuid()}",
    $"{Guid.NewGuid()}",
    "Road Bike",
    500,
    "rd-bk-500"
);

ItemResponse<Product> response = await container.CreateItemAsync<Product>(item);

Product createdItem = response.Resource;

Console.WriteLine($"RUs:\t{response.RequestCharge:0.00}");
```

Sample console output

```
RUs: 7.05
```

Lab – Optimize an Azure Cosmos DB for NoSQL container's index policy for a specific query



- Create an Azure Cosmos DB for NoSQL account
- Seed your Azure Cosmos DB for NoSQL account with sample data
- Execute SQL queries and measure their request unit charge
- Create a composite index in the indexing policy

Implement integrated cache

Review workloads that benefit from the cache

Workloads that consistently perform the same point read and query operations are ideal to use with the integrated cache.

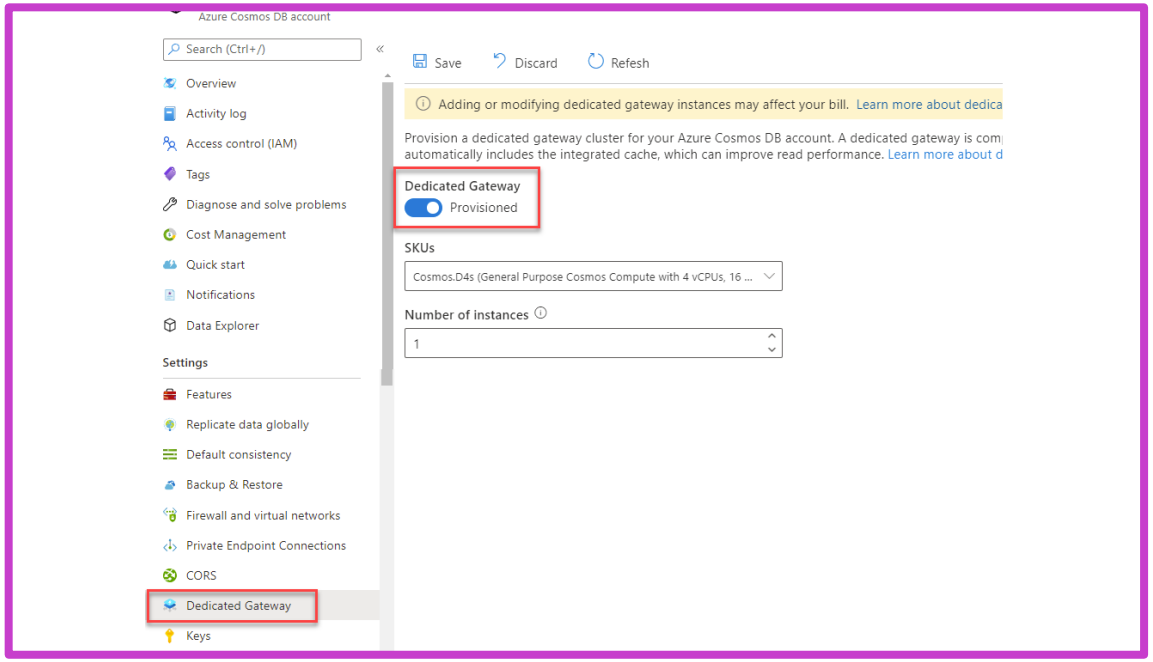
Workloads in Azure Cosmos DB that can benefit form the integrated cache:

- Workloads with far more read operations and queries than write operations
- Workloads that read large individual items multiple times
- Workloads that execute queries multiple times with a large amount of RU/s
- Workloads that have hot partition key[s] for read operations and queries

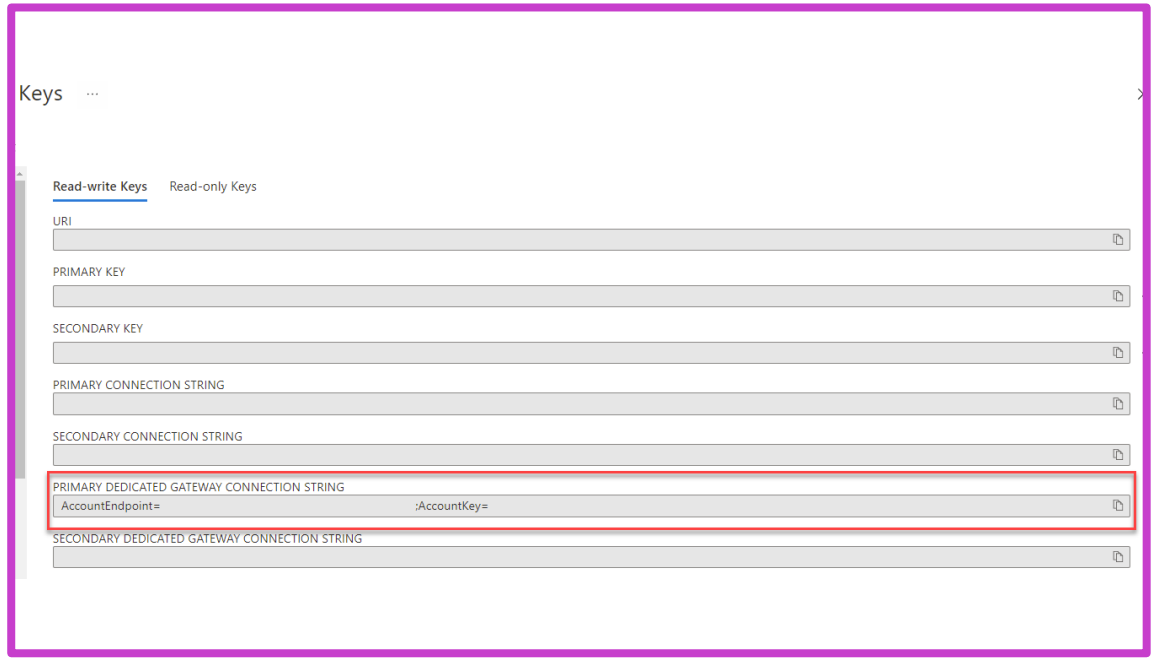
Enable integrated cache – Create a dedicated gateway

First step, Create a dedicated gateway in your Azure Cosmos DB for NoSQL account.

Create a dedicated gateway



Get the connection string for the gateway



Enable integrated cache – Update .NET SDK code

For the .NET SDK client to use the integrated cache you need the following changes:

The client uses the *dedicated gateway connection string* instead of the typical connection string

```
// For the dedicated gateway, the connection string is in the structure of <cosmos-account-name>.sqlx.cosmos.azure.com.
string connectionString = "AccountEndpoint=https://<cosmos-account-name>.sqlx.cosmos.azure.com/;AccountKey=<cosmos-key>;";
```

The client is configured to use *Gateway* mode instead of the default *Direct* connectivity mode

```
// Set the ConnectionMode property of the CosmosClientOptions class to ConnectionMode.Gateway.
CosmosClientOptions options = new()
{
    ConnectionMode = ConnectionMode.Gateway
};

CosmosClient client = new (connectionString, options);
```

The client's *consistency level* must be set to *session* or *eventual*

```
string sql = "SELECT * FROM products";
QueryDefinition query = new(sql);

// Set the ConsistencyLevel property of the QueryRequestOptions class to ConsistencyLevel.Session or ConsistencyLevel.Eventual.
QueryRequestOptions queryOptions = new()
{
   ConsistencyLevel = ConsistencyLevel.Eventual
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOptions: queryOptions);
```

Configure cache staleness

By default, the cache will keep data for five minutes. This staleness window can be configured using the *MaxIntegratedCacheStaleness* property in the SDK.

```
ItemRequestOptions operationOptions = new()
{
    ConsistencyLevel = ConsistencyLevel.Eventual,
    DedicatedGatewayRequestOptions = new()
    {
        MaxIntegratedCacheStaleness = TimeSpan.FromMinutes(15)
    }
};
```

```
QueryRequestOptions queryOptions = new()
{
    ConsistencyLevel = ConsistencyLevel.Eventual,
    DedicatedGatewayRequestOptions = new()
    {
        MaxIntegratedCacheStaleness = TimeSpan.FromSeconds(120)
    }
};
```