Module 10: Optimize query and operation performance in Azure Cosmos DB SQL API

In [ ]:

```
using Microsoft.Azure.Cosmos;
using System;
using System.Collections.Generic;

CosmosClient client = new (connectionString);
Database database = client.GetDatabase("cosmicworks");
Container container = database.GetContainer("products");

public class Product
{
        public string id { get; set; }
        public string categoryId { get; set; }
        public string categoryName { get; set; }
        public string sku { get; set; }
        public string name { get; set; }
        public string description { get; set; }
        public double price { get; set; }
}
```

Optimize indexes in Azure Cosmos DB SQL API

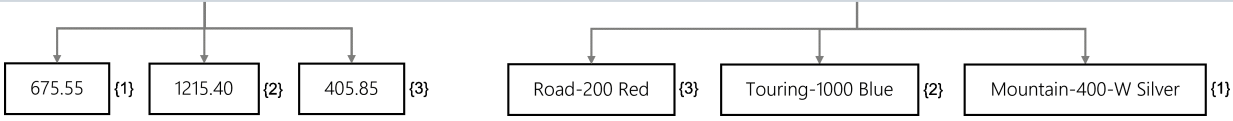
Index usage

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

Method	Description	RU implication
Index seek	The query engine will seek an exact match on a field's value by traversing directly to that value and looking up how many items match. Once the matched items are determined, the query engine will return the items as the query result.	The RU charge is constant for the lookup. The RU charge for loading and returning items is linear based on the number of items.
Index scan	The query engine will find all possible values for a field and then perform various comparisons only on the values. Once matches are found, the query engine will load and return the items as the query result.	The RU charge is still constant for the lookup, with a slight increase over the index seek based on the cardinality of the indexed properties. The RU charge for loading and returning items is still linear based on the number of items returned.
Full scan	The query engine will load the items, in their entirety, to the transactional store to evaluate the filters.	This type of scan does not use the index; however, the RU charge for loading items is based on the number of items in the entire container.

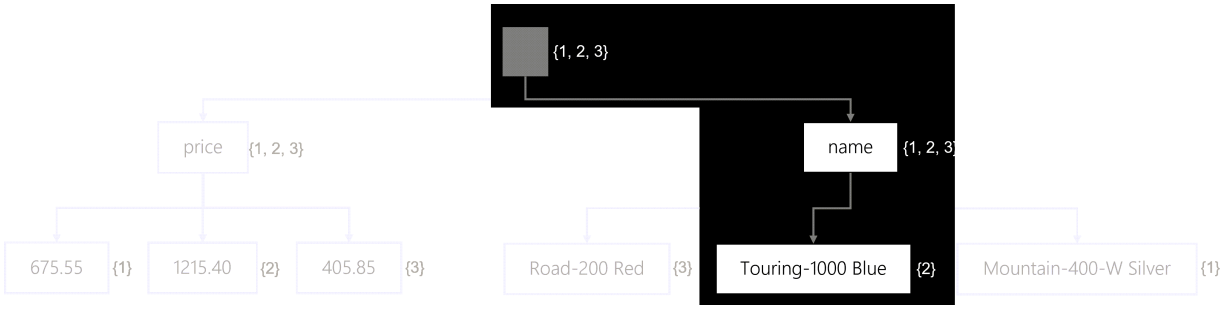
Suppose the items in the product container are:

```
[
  { "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 }
]
```

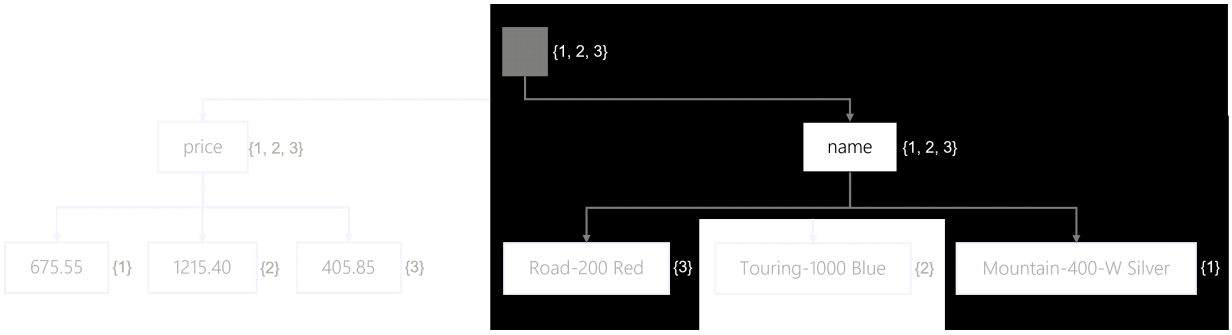


How will the index be used with the following queries?

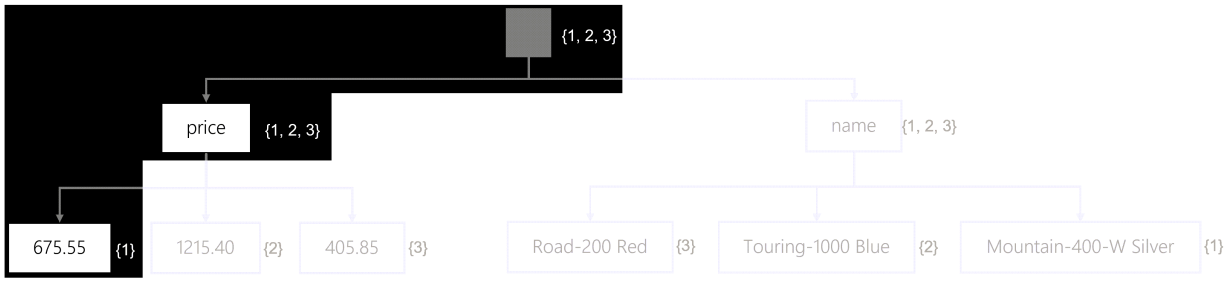
```
SELECT *
FROM products p
WHERE p.name = 'Touring-1000 Blue'
```




```
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 <= 1000
```



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 indexing policy:

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

Proposed indexing 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" ]
}
```

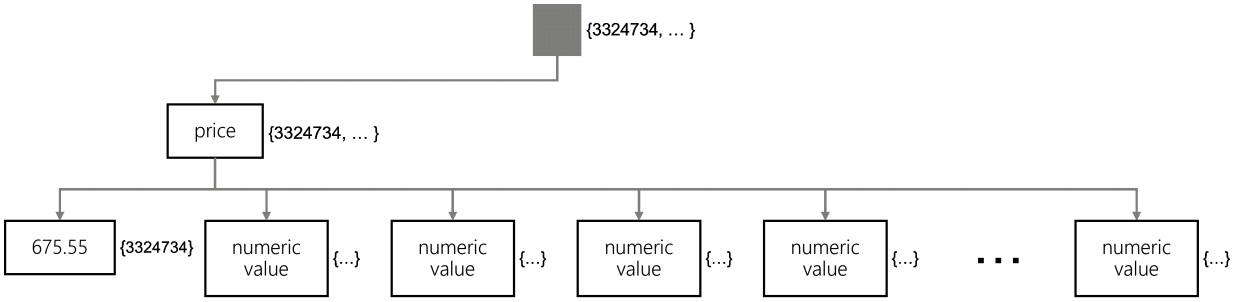
Assume the application only uses these two queries

```
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": "/*" } ]
}
```



Suppose your application is write-heavy and only ever does point reads using the id and partition key values. In that case, you can choose to disable indexing entirely using a customized indexing policy.

```
{
  "indexingMode": "none",
}
```

Measure index performance in Azure Cosmos DB SQL API

Enable indexing metrics

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

In [ ]:

```
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 perform
QueryRequestOptions options = new()
{
   PopulateIndexMetrics = true
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOpti

while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {
        product.price++;
```

```
        //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 ou
    Console.WriteLine(response.IndexMetrics);
}
```

Analyze indexing metrics results

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

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

In [ ]:

```
QueryDefinition query = new("SELECT * FROM products p WHERE p.price > 500");

//  PopulateIndexMetrics is disabled by default, enable it if troubleshooting query perform
QueryRequestOptions options = new()
{
   PopulateIndexMetrics = true
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOpti

while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {
        product.price++;
        //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 ou
    Console.WriteLine(response.IndexMetrics);
}
```

Another query:

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

In [ ]:

```
QueryDefinition query = new("SELECT * FROM products p WHERE p.price > 500 AND startsWith(p.

//  PopulateIndexMetrics is disabled by default, enable it if troubleshooting query perform
QueryRequestOptions options = new()
{
   PopulateIndexMetrics = true
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOpti

while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {
        product.price++;
        //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 ou
    Console.WriteLine(response.IndexMetrics);
}
```

```
}
```

Analyze indexing metrics results – composite indexes

The indexing metrics could recommend we create a composite index.

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

In [ ]:

```
QueryDefinition query = new("SELECT * FROM products p WHERE p.price > 500 AND p.categoryNam

//  PopulateIndexMetrics is disabled by default, enable it if troubleshooting query perform
QueryRequestOptions options = new()
{
   PopulateIndexMetrics = true
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOpti

while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {
        product.price++;
        //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 ou
    Console.WriteLine(response.IndexMetrics);
}
```

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" }
  ] ]
}
```

In [ ]:

```
// Add a composite index
using System.Collections.ObjectModel;

IndexingPolicy policy = new ()
{
    IndexingMode = IndexingMode.Consistent,
    Automatic = true
};

policy.IncludedPaths.Add( new IncludedPath{ Path = "/*" } );
policy.ExcludedPaths.Add( new ExcludedPath{ Path = "/_etag/?" } );
policy.CompositeIndexes.Add(new Collection<CompositePath> {
    new CompositePath() { Path = "/categoryName", Order = CompositePathSortOrder.Ascending
    new CompositePath() { Path = "/price", Order = CompositePathSortOrder.Ascending }
});

ContainerProperties options = new () {
    Id = "products",
    PartitionKeyPath = "/categoryId",
    IndexingPolicy = policy };
```

```
await container.ReplaceContainerAsync(options);
```

Run the query one more time and validate it the **"Utilized Composite Indexes"**

In [ ]:

```
QueryDefinition query = new("SELECT * FROM products p WHERE p.price > 500 AND p.categoryNam

//  PopulateIndexMetrics is disabled by default, enable it if troubleshooting query perform
QueryRequestOptions options = new()
{
   PopulateIndexMetrics = true
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOpti

while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {
        product.price++;
        //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 ou
    Console.WriteLine(response.IndexMetrics);
}
```

Measure query cost

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

In [ ]:

```
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
QueryRequestOptions options = new()
{
   MaxItemCount = 25
};

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOpti

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}");
```

Measure point operation cost

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

In [ ]:

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

```
Product item = new()
{
    id = $"{Guid.NewGuid()}",
    categoryId = "26C74104-40BC-4541-8EF5-9892F7F03D72",
    name = "LL Road Seat/Saddle",
    price = 27.12d
};

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

Product createdItem = response.Resource;

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

Implement integrated cache

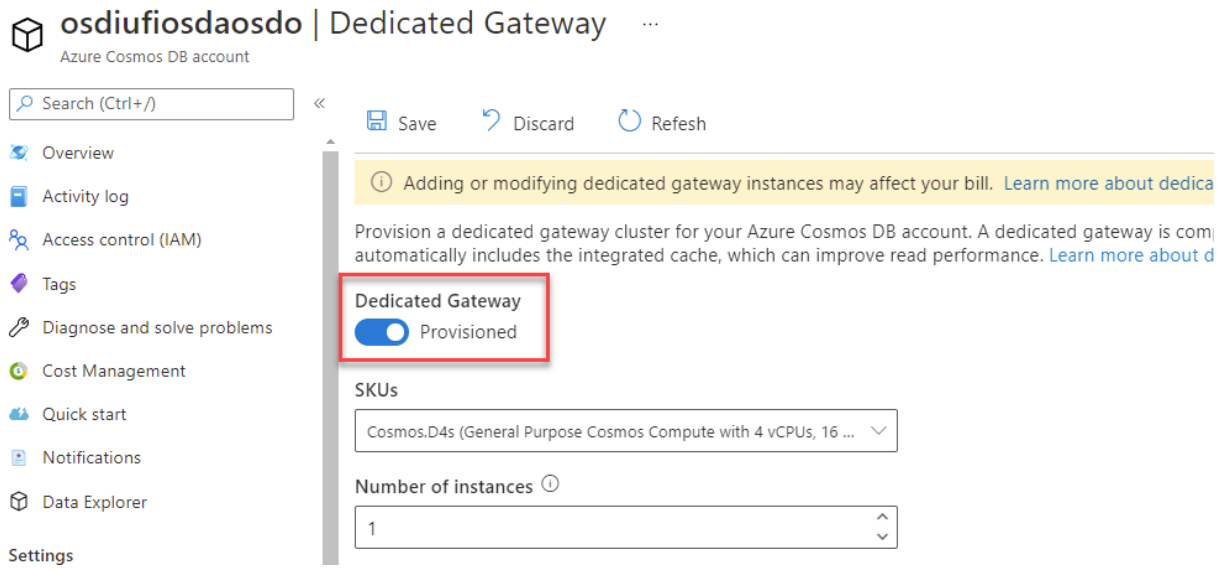
Review workloads that benefit from the cache

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

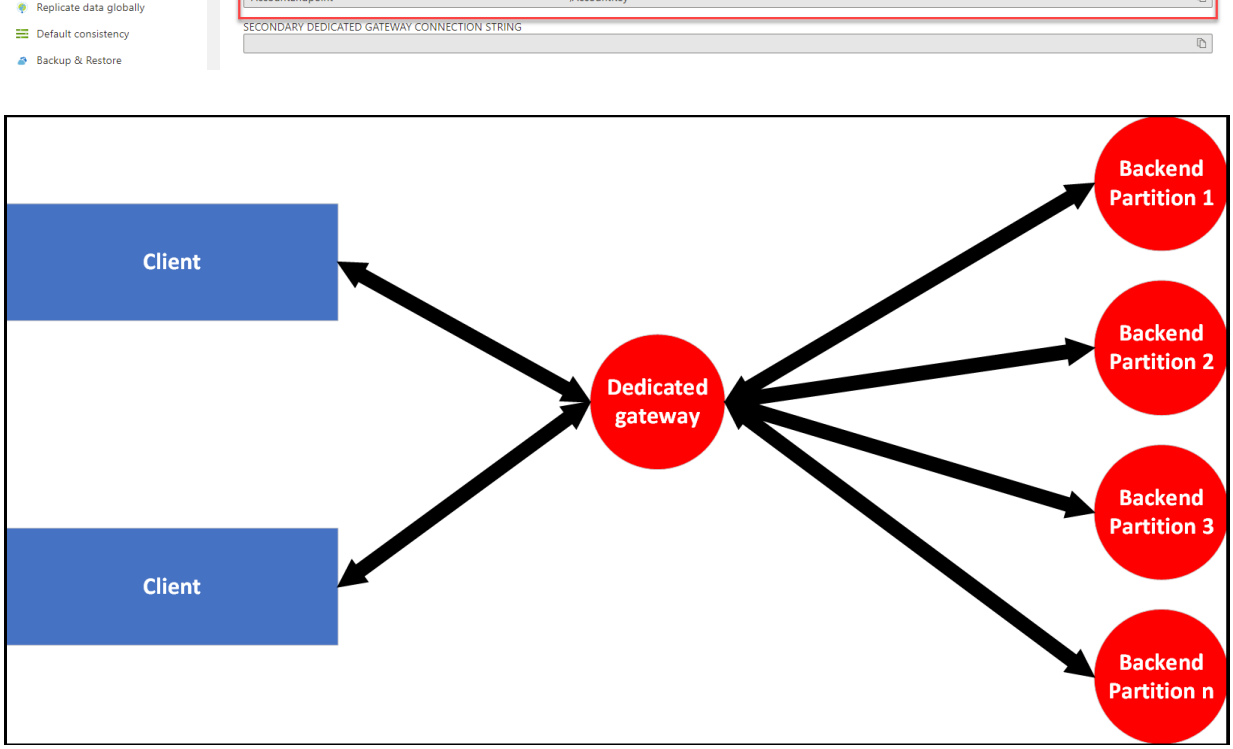
- Workloads with far more read operations and queries than write operations
- Workloads that read large individual items multiple times
- Workloads that execute queries multiple times with a large amount of RU/s
- Workloads that have hot partition key[s] for read operations and queries

Enable integrated cache – Create a dedicated gateway

First step, Create a dedicated gateway in your Azure Cosmos DB SQL API account.



Get the connection string for the gateway



The dedicated gateway has the following limitations during the public preview:

- Dedicated gateways are only supported on SQL API accounts.
- You can't provision a dedicated gateway in Azure Cosmos DB accounts with IP firewalls or Private Link configured.
- You can't provision a dedicated gateway in an Azure Cosmos DB account in a Virtual Network (Vnet)
- You can't provision a dedicated gateway in Azure Cosmos DB accounts with availability zones.
- You can't use role-based access control (RBAC) to authenticate data plane requests routed through the dedicated 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

("AccountEndpoint=https://.sqlx.cosmos.azure.com/;AccountKey=;")

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

The integrated cache has two parts:

- An item cache for point reads
- A query cache for queries

In [ ]:
```
$dedicatedConnectionString = (Get-AzCosmosDBAccountKey -ResourceGroupName rg-dp-420 -Name c
$dedicatedConnectionString
```

In [ ]:
```
#!share --from pwsh dedicatedConnectionString

// The client is configured to use Gateway mode instead of the default Direct connectivity
CosmosClientOptions options = new()
{
    ConnectionMode = ConnectionMode.Gateway
};

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

```
// The client's consistency level must be set to session or eventual
QueryDefinition query = new("SELECT * FROM products");

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

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOpti

double totalRUs = 0;

while(iterator.HasMoreResults)
{
    FeedResponse<Product> response = await iterator.ReadNextAsync();
    foreach(Product product in response)
    {
        // Do something with each product
        product.price++;
    }
    //  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}");
```

Finally, you can restart your application and verify integrated cache hits for repeated point reads or queries.

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.

Note: Customizing MaxIntegratedCacheStaleness is only supported in the latest .NET and Java preview SDK's.

In [ ]:

```
QueryRequestOptions queryOptions = new()
{
    ConsistencyLevel = ConsistencyLevel.Eventual,

    DedicatedGatewayRequestOptions = new()
    {
        MaxIntegratedCacheStaleness = TimeSpan.FromSeconds(20)
    }
};

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

FeedIterator<Product> iterator = container.GetItemQueryIterator<Product>(query, requestOpti

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;
}
```