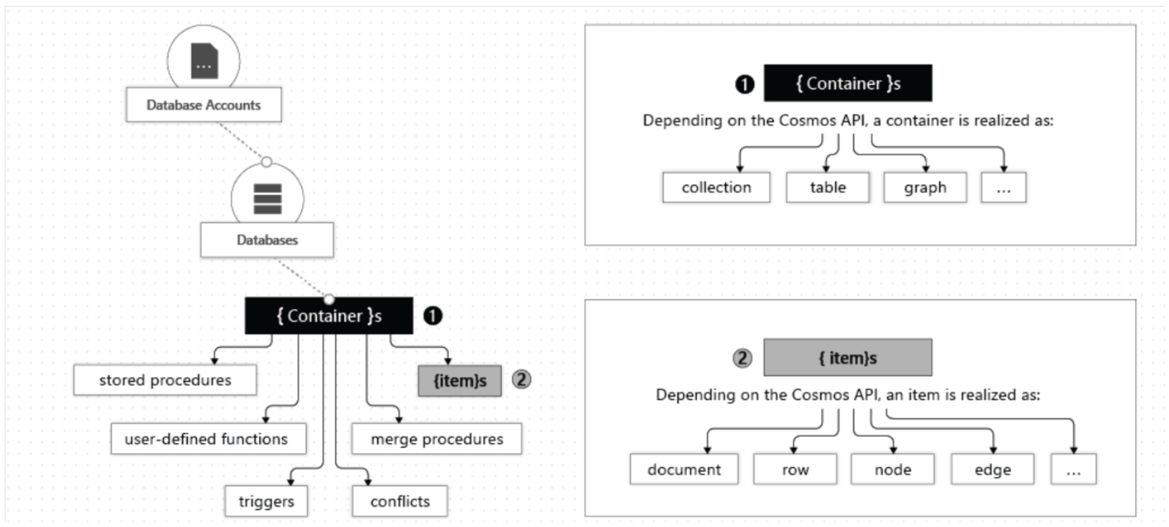
II.1 Develop solutions that use Cosmos DB storage

jeudi 16 septembre 2021 14:24

Resources hierarchy



- An Azure Cosmos container is the unit of scalability both for provisioned throughput and storage.
- A container is horizontally partitioned and then replicated across multiple regions.
- The items that you add to the container are automatically grouped into logical partitions, which are distributed across physical partitions, based on the partition key.
- The throughput on a container is evenly distributed across the physical partitions.

- When you create a container, you configure **throughput** in one of the following modes:
  - **Dedicated** provisioned throughput mode: The throughput provisioned on a container is exclusively reserved for that container and it is backed by the SLAs.
  - **Shared** provisioned throughput mode: These containers share the provisioned throughput with the other containers in the same database (excluding containers been configured with dedicated provisioned throughput). In other words, the provisioned throughput on the database is shared among all the "shared throughp containers.

- A container is a schema-agnostic container of items.

- Items in a container can have arbitrary schemas. For example, an item that represents a person and an item that represents an automobile can be placed in the same

- By default, all items that you add to a container are automatically indexed without requiring explicit index or schema management.

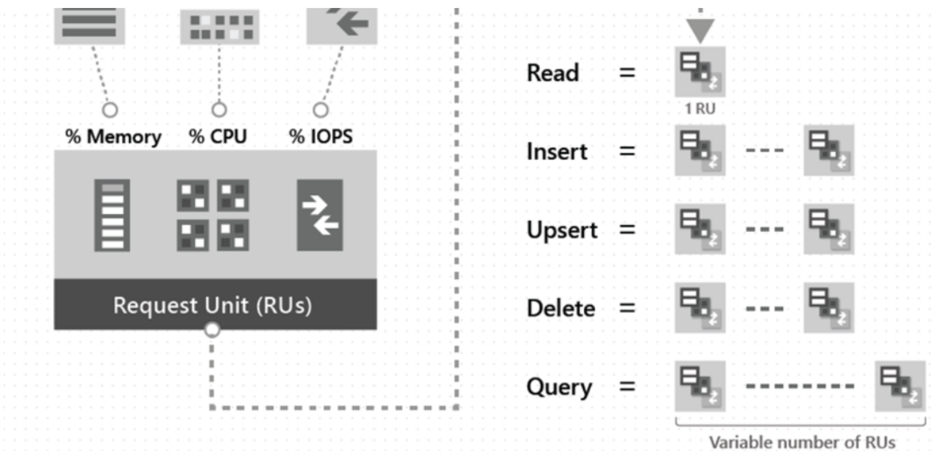
Select the appropriate API and SDK for a solution

For product catalog data, Azure Cosmos DB is the preferred choice :
- Supports semi-structured data, json: flexibility in schema.
- Support sql for queries.
- ACID Compliant.
- Replication.
- Fine-grained cost management with the ability to choose consistency levels to determine trade-offs between consistency, availability, latency, and throughput.

- **Core(SQL) API** :
  - Stores data in document format
  - Query using SQL syntax
  - Recommended option when migrating from other databases.
- **Api for MongoDB**
  - Stores data in document structure
  - Use with MongoDB ecosystem
- **Cassandra API**
  - Stores data in column-oriented schema
  - Use with Cassandra ecosystem
- **Table API**
  - Stores data in key/value format
  - When you want to migrate from Azure Table storage for better latency, scaling, throughput and performance
- **Gremlin API**
  - For Graph queries, stores data as edges and vertices
  - Use with dynamic data with complex relations
  - Use with Gremlin ecosystem

Request Units

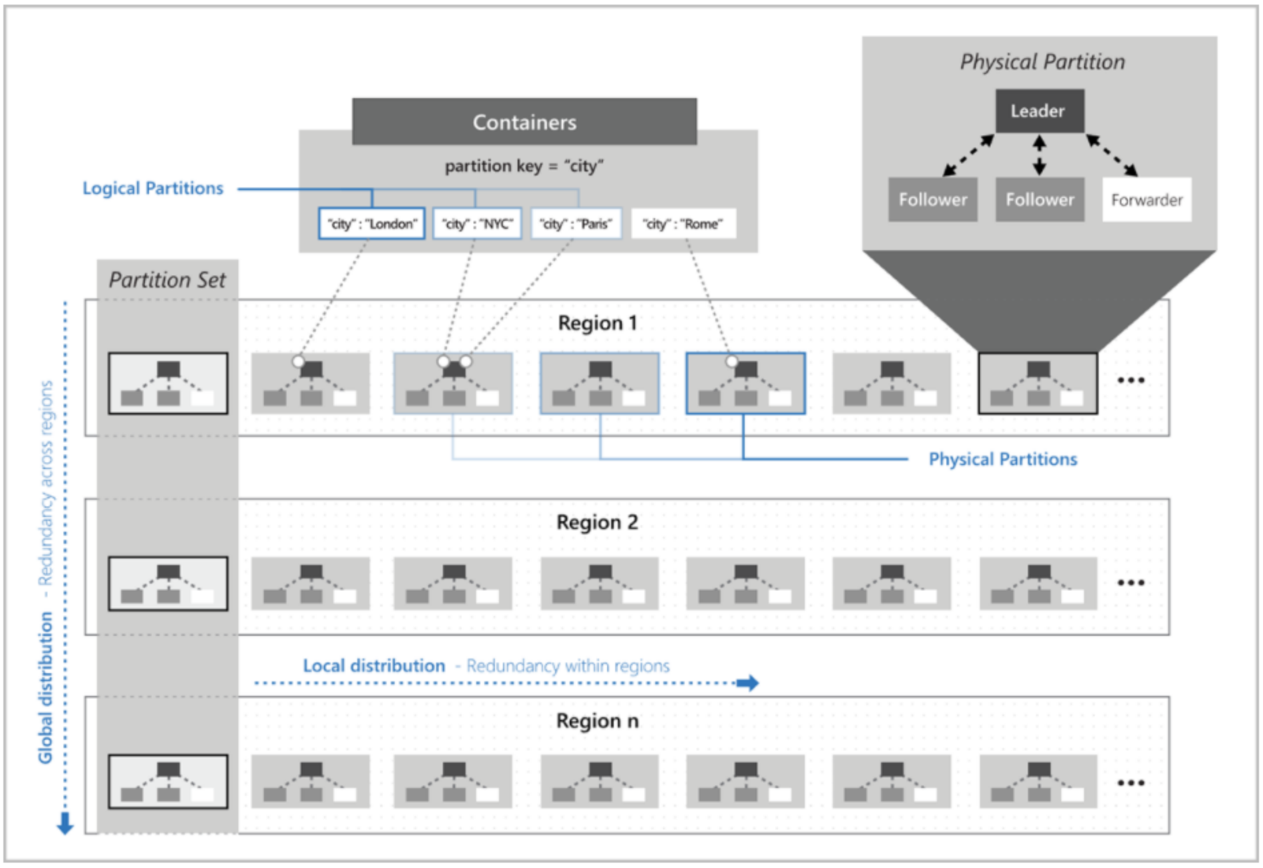
Usage is expressed in Request Units
Database operations consume a variable number of RUs



When creating an Azure Cosmos account :

- **Provisioned throughput mode** : provision the number of RUs for your application on a per-second basis in increments of 100 RUs per second.
- **Serverless mode** : At the end of your billing period, you get billed for the amount of request units that has been consumed by your database operations.
- **Autoscale mode :** You can automatically and instantly scale the throughput (RU/s) of your database or container based on it's usage.

Implement partitioning schemes and partition keys



Partition key = path + value (ex: /"userId" + "Andrew")

It must be :

- Non updatable
- Large field of possible values (no duplicates)

Perform operations on data and Cosmos DB containers

SQL requests examples

```
SELECT {"Name":f.id, "City":f.address.city} AS Family
    FROM Families f
    WHERE f.address.city = f.address.state

SELECT c.givenName
    FROM Families f
    JOIN c IN f.children
    WHERE f.id = 'WakefieldFamily'
    ORDER BY f.address.city ASC
```

Create Cosmos in Azure

```
# Set variables for the new SQL API account, database, and container
resourceGroupName='myResourceGroup'
location='southcentralus'

# The Azure Cosmos account name must be globally unique, so be sure to update the `mysqlapicosmosdb` value before y
accountName='mysqlapicosmosdb'

# Create a resource group
az group create \
    --name $resourceGroupName \
    --location $location

# Create a SQL API Cosmos DB account with session consistency and multi-region writes enabled
az cosmosdb create \
    --resource-group $resourceGroupName \
    --name $accountName \
    --kind GlobalDocumentDB \
    --locations regionName="South Central US" failoverPriority=0 --locations regionName="North Central US" failover
    --default-consistency-level "Session" \
    --enable-multiple-write-locations true
```

Create .NET app

dotnet new console --langVersion:8 -n todo
dotnet add package Azure.Cosmos --version 4.0.0-preview3

Resource hierarchy in CosmosDB: Azure Cosmos Account > Databases > Containers > Items

Classes & methods in .NET :

- CosmosClient : base class
- CreateDatabaseIfNotExistsAsync, CreateContainerIfNotExistsAsync, CreateItemAsync

, UpsertItemAsync, GetItemQueryIterator, DeleteAsync

Using : using Azure.Cosmos;

Database

Create a database

The `CosmosClient.CreateDatabaseIfNotExistsAsync` checks if a database exists, and if it doesn't, creates it. Only the database `id` is used to verify if there is an existing database.

```C#
// An object containing relevant information about the response
DatabaseResponse databaseResponse = await client.CreateDatabaseIfNotExistsAsync(databaseId, 10000);
```

Read a database by ID

Reads a database from the Azure Cosmos service as an asynchronous operation.

```C#
DatabaseResponse readResponse = await database.ReadAsync();
```

Delete a database

Delete a Database as an asynchronous operation.

```C#
await database.DeleteAsync();
```

Containers

Create a container

The `Database.CreateContainerIfNotExistsAsync` method checks if a container exists, and if it doesn't, it creates it. Only the container `id` is used to verify if there is an existing container.

C# Copy

```csharp
// Set throughput to the minimum value of 400 RU/s
ContainerResponse simpleContainer = await database.CreateContainerIfNotExistsAsync(
    id: containerId,
    partitionKeyPath: partitionKey,
    throughput: 400);
```

Get a container by ID

C# Copy

```csharp
Container container = database.GetContainer(containerId);
ContainerProperties containerProperties = await container.ReadContainerAsync();
```

Delete a container

Delete a Container as an asynchronous operation.

C# Copy

```csharp
await database.GetContainer(containerId).DeleteContainerAsync();
```

Items

Create an item

Use the `Container.CreateItemAsync` method to create an item. The method requires a JSON serializable object that must contain an `id` property, and a `partitionKey`.

C# Copy

```csharp
ItemResponse<SalesOrder> response = await container.CreateItemAsync(salesOrder, new PartitionKey(salesOrder.Ac
```

Read an item

Use the `Container.ReadItemAsync` method to read an item. The method requires type to serialize the item to along with an `id` property, and a `partitionKey`.

C# Copy

```csharp
string id = "[id]";
string accountNumber = "[partition-key]";
ItemResponse<SalesOrder> response = await container.ReadItemAsync(id, new PartitionKey(accountNumber));
```

Query an item

The `Container.GetItemQueryIterator` method creates a query for items under a container in an Azure Cosmos database using a SQL statement with parameterized values. It returns a `FeedIterator`.

C# Copy

```csharp
QueryDefinition query = new QueryDefinition(
    "select * from sales s where s.AccountNumber = @AccountInput ")
    .WithParameter("@AccountInput", "Account1");

FeedIterator<SalesOrder> resultSet = container.GetItemQueryIterator<SalesOrder>(
    query,
    requestOptions: new QueryRequestOptions()
    {
        PartitionKey = new PartitionKey("Account1"),
        MaxItemCount = 1
    });
```

Use the feedIterator

```csharp
using (FeedIterator feedIterator = this.Container.GetItemQueryStreamIterator(
    queryDefinition))
{
    while (feedIterator.HasMoreResults)
    {
        // Stream iterator returns a response with status code
        using(ResponseMessage response = await feedIterator.ReadNextAsync())
        {
            // Handle failure scenario
            // handle failure scenario
            if(!response.IsSuccessStatusCode)
            {
                // Log the response.Diagnostics and handle the error
            }
        }
    }
}
```

Stored procedures : Register and call

You cannot write triggers, stored procedures, and user-defined functions (UDFs) in a Cosmos DB database running Table API. You can write code with custom business logic in Javascript that is executed inside the database engine using the SQL API only. Other Cosmos DB APIs do not support triggers, stored procedures, and UDFs.

You cannot write stored procedures in Cosmos DB using C# code. Cosmos DB only supports stored procedures, triggers, and UDFs written in Javascript. You can register and use Cosmos DB server-side programming custom logic with supported SDKs, including C#, Java, Javascript, and Python, but the code that is executed by the database engine must be written in Javascript.

```
// Register
string storedProcedureId = "spCreateToDoItems";
StoredProcedureResponse storedProcedureResponse = await client.GetContainer("myDatabase", "myContainer")
    .Scripts
    .CreateStoredProcedureAsync(new StoredProcedureProperties
{
    Id = storedProcedureId,
    Body = File.ReadAllText($@"..\js\{storedProcedureId}.js")
});

// Call
dynamic[] newItems = new dynamic[]
{
    new {
        category = "Personal",
        name = "Groceries",
        description = "Pick up strawberries",
        isComplete = false
    },
    new {
        category = "Personal",
        name = "Doctor",
        description = "Make appointment for check up",
        isComplete = false
    }
};

var result = await client.GetContainer("database", "container")
    .Scripts
    .ExecuteStoredProcedureAsync<string>("spCreateToDoItem", new PartitionKey("Personal"), new[] { newItems });
```

Pre-trigger : Register and call

```
// Register
await client.GetContainer("database", "container").Scripts.CreateTriggerAsync(new TriggerProperties
{
    Id = "trgPreValidateToDoItemTimestamp",
    Body = File.ReadAllText("@..\js\trgPreValidateToDoItemTimestamp.js"),
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Pre
});

// Call
dynamic newItem = new
{
    category = "Personal",
    name = "Groceries",
    description = "Pick up strawberries",
    isComplete = false
};

await client.GetContainer("database", "container")
.CreateItemAsync(newItem, null, new ItemRequestOptions { PreTriggers = new List<string> { "trgPreValidateToDoItemTimestamp" } });
```

Post-trigger : Register and call

```
// Register
await client.GetContainer("database", "container").Scripts.CreateTriggerAsync(new TriggerProperties
{
    Id = "trgPostUpdateMetadata",
    Body = File.ReadAllText(@"..\js\trgPostUpdateMetadata.js"),
    TriggerOperation = TriggerOperation.Create,
    TriggerType = TriggerType.Post
});

// Call
var newItem = {
    name: "artist_profile_1023",
    artist: "The Band",
    albums: ["Hellujah", "Rotators", "Spinning Top"]
};

await client.GetContainer("database", "container")
.CreateItemAsync(newItem, null, new ItemRequestOptions { PostTriggers = new List<string> { "trgPostUpdateMetadata" } });
```

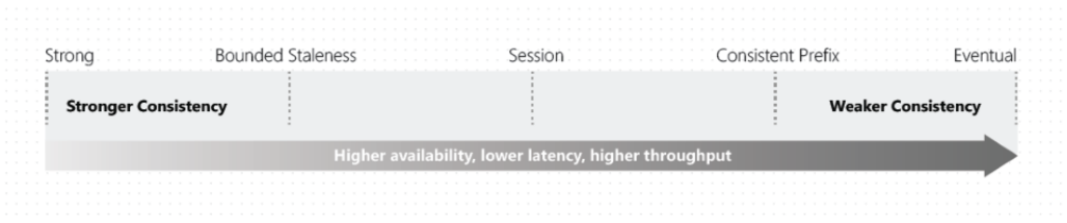
User-defined functions

```
// Register
await client.GetContainer("database", "container")
    .Scripts
    .CreateUserDefinedFunctionAsync(new UserDefinedFunctionProperties
{
    Id = "Tax",
    Body = File.ReadAllText(@"..\js\Tax.js")
});

// Call
var iterator = client.GetContainer("database", "container")
.GetItemQueryIterator<dynamic>("SELECT * FROM Incomes t WHERE udf.Tax(t.income) > 20000");

while (iterator.HasMoreResults)
{
    var results = await iterator.ReadNextAsync();
    foreach (var result in results)
    {
        //iterate over results
    }
}
```

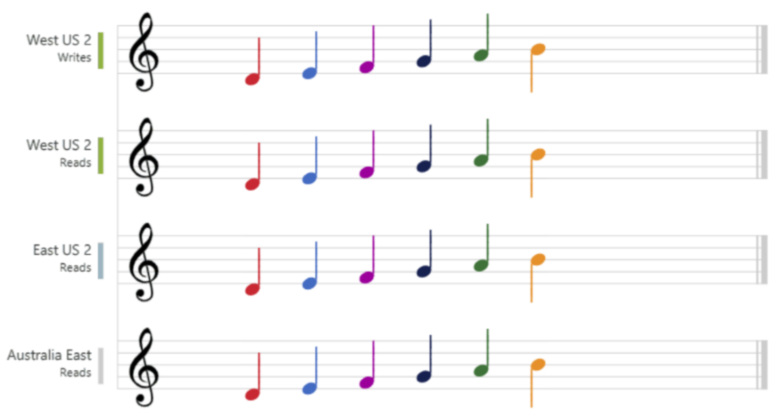
Set the appropriate consistency level for operations



Levels	Consistency	Availability	Latency	Throughput
Strong	Highest	Low	High	Low
Bounded Staleness	High	Low	High	Low
Session	Moderate	High	Moderate	Moderate
Consistent Prefix	Low	High	Low	Moderate
Eventual	Low	High	Low	High

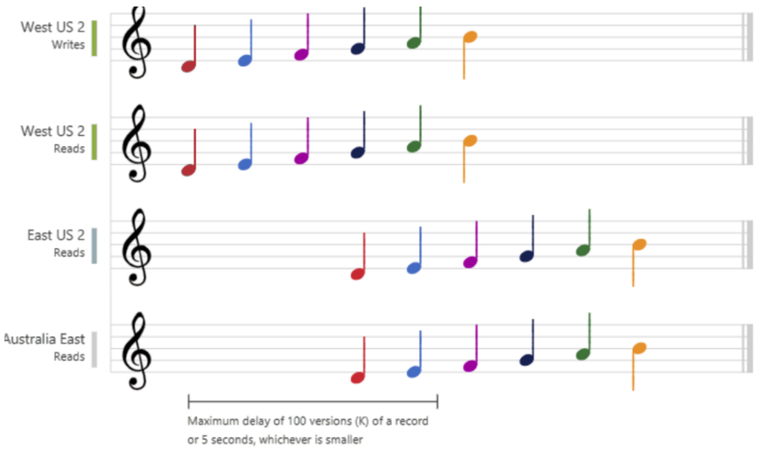
1. Strong

Reads are guaranteed to see the most recent write



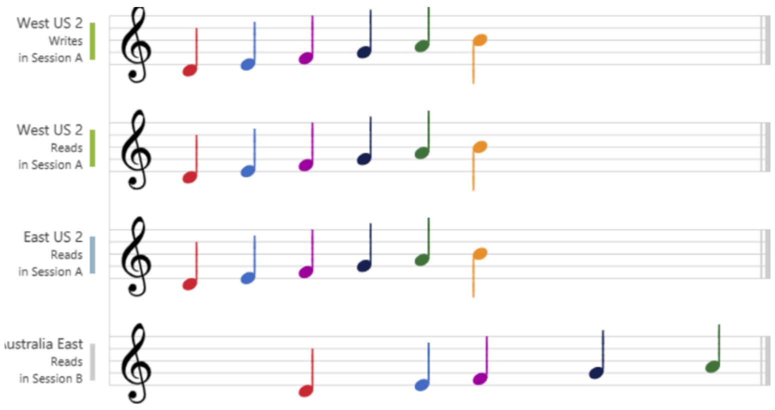
Bounded staleness

Bounded staleness is frequently chosen by globally distributed applications that expect low write latencies but require total global order guarante is introduced before information is available on all regions. This consistency level includes implementation of consistent prefix, whch guarantees tl writes are returned in order. You can configure staleness, which determines the data version returned, based on either the number of versions or c delay.


West US 2
Writes
West US 2
Reads
East US 2
Reads
Australia East
Reads
Maximum delay of 100 versions (K) of a record
or 5 seconds, whichever is smaller

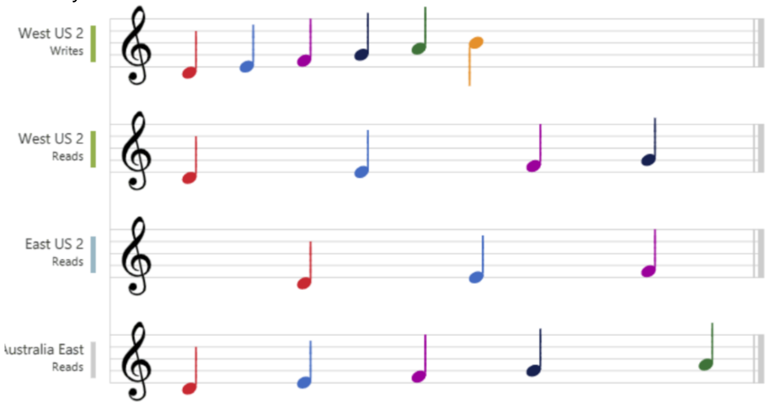
Session

Session consistency is the most widely used consistency level for both single region as well as globally distributed applications. This is the default consistency model. Only support a single client session. You can't know data version (staleness).



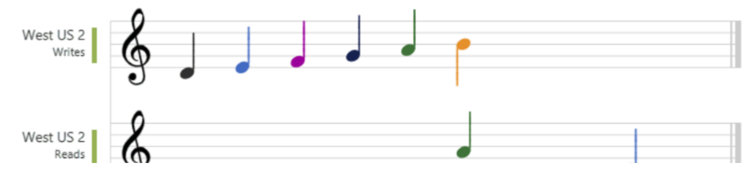
Consistent prefix

Consistent Prefix provides write latencies, availability, and read throughput comparable to that of eventual consistency, but also provides the orde guarantees that suit the needs of scenarios where order is important. Is guarantees that reads never see out of order writes. You have no staleness means you can't know the data version returned.



Eventual

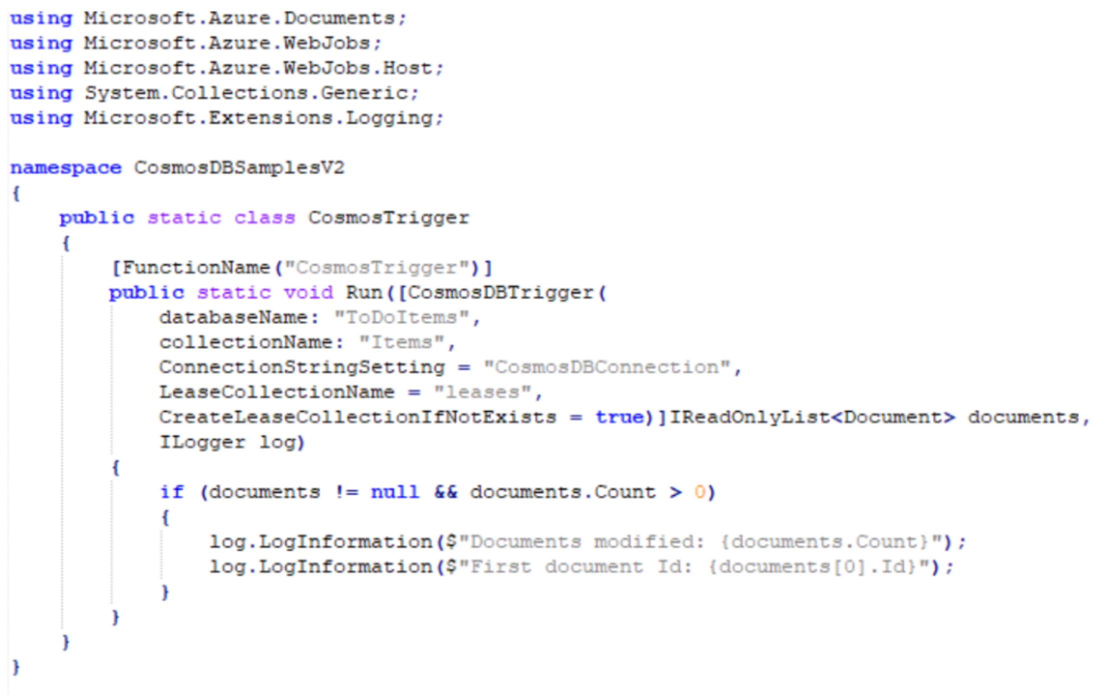
In eventual consistency, there's no ordering guarantee for reads. In the absence of any further writes, the replicas eventually converge.
Eventual consistency is the weakest form of consistency because a client may read the values that are older than the ones it had read before. Even consistency is ideal where the application does not require any ordering guarantees.



East US 2
Reads
Australia East
Reads

Manage change feed notifications

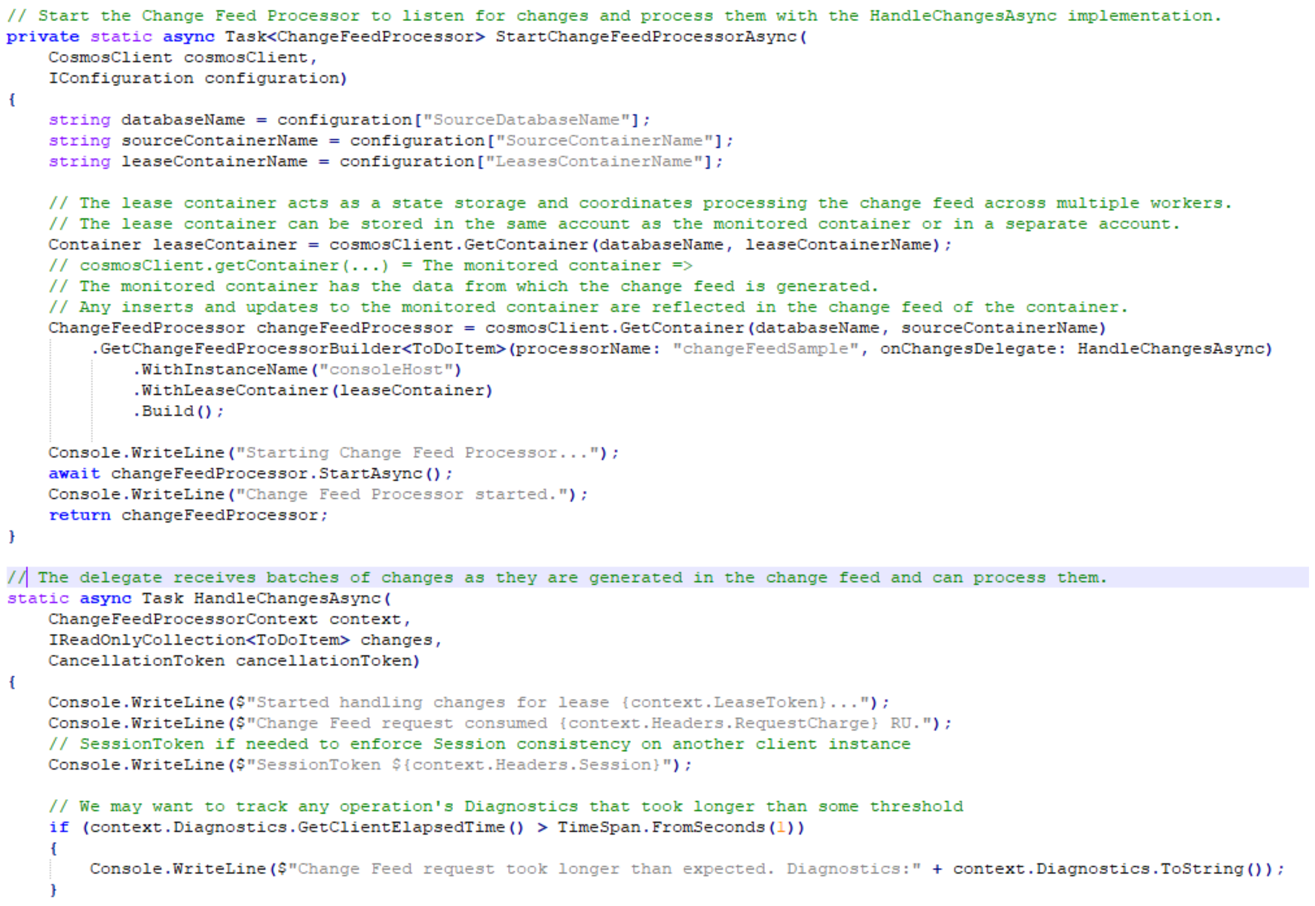
1. With Azure function



```
using Microsoft.Azure.Documents;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Host;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;

namespace CosmosDBSamplesV2
{
    public static class CosmosTrigger
    {
        [FunctionName("CosmosTrigger")]
        public static void Run([CosmosDBTrigger(
            databaseName: "ToDoItems",
            collectionName: "Items",
            ConnectionStringSetting = "CosmosDBConnection",
            LeaseCollectionName = "leases",
            CreateLeaseCollectionIfNotExists = true)]IReadOnlyList<Document> documents,
            ILogger log)
        {
            if (documents != null && documents.Count > 0)
            {
                log.LogInformation($"Documents modified: {documents.Count}");
                log.LogInformation($"First document Id: {documents[0].Id}");
            }
        }
    }
}
```

2. With ChangeFeedProcessor



```
// Start the Change Feed Processor to listen for changes and process them with the HandleChangesAsync implementation.
private static async Task<ChangeFeedProcessor> StartChangeFeedProcessorAsync(
    CosmosClient cosmosClient,
    IConfiguration configuration)
{
    string databaseName = configuration["SourceDatabaseName"];
    string sourceContainerName = configuration["SourceContainerName"];
    string leaseContainerName = configuration["LeasesContainerName"];

    // The lease container acts as a state storage and coordinates processing the change feed across multiple workers.
    // The lease container can be stored in the same account as the monitored container or in a separate account.
    Container leaseContainer = cosmosClient.GetContainer(databaseName, leaseContainerName);
    // cosmosClient.getContainer(...) = The monitored container =>
    // The monitored container has the data from which the change feed is generated.
    // Any inserts and updates to the monitored container are reflected in the change feed of the container.
    ChangeFeedProcessor changeFeedProcessor = cosmosClient.GetContainer(databaseName, sourceContainerName)
        .GetChangeFeedProcessorBuilder<ToDoItem>(processorName: "changeFeedSample", onChangesDelegate: HandleChangesAsync)
            .WithInstanceName("consoleHost")
            .WithLeaseContainer(leaseContainer)
            .Build();

    Console.WriteLine("Starting Change Feed Processor...");
    await changeFeedProcessor.StartAsync();
    Console.WriteLine("Change Feed Processor started.");
    return changeFeedProcessor;
}

// The delegate receives batches of changes as they are generated in the change feed and can process them.
static async Task HandleChangesAsync(
    ChangeFeedProcessorContext context,
    IReadOnlyCollection<ToDoItem> changes,
    CancellationToken cancellationToken)
{
    Console.WriteLine($"Started handling changes for lease {context.LeaseToken}...");
    Console.WriteLine($"Change Feed request consumed {context.Headers.RequestCharge} RU.");
    // SessionToken if needed to enforce Session consistency on another client instance
    Console.WriteLine($"SessionToken ${context.Headers.Session}");

    // We may want to track any operation's Diagnostics that took longer than some threshold
    if (context.Diagnostics.GetClientElapsedTime() > TimeSpan.FromSeconds(1))
    {
        Console.WriteLine($"Change Feed request took longer than expected. Diagnostics:" + context.Diagnostics.ToString());
    }
    foreach (ToDoItem item in changes)
    {
        Console.WriteLine($"Detected operation for item with id {item.id}, created at {item.creationTime}.");
        // Simulate some asynchronous operation
        await Task.Delay(10);
    }

    Console.WriteLine("Finished handling changes.");
}
```