ntre muchas de las características que hacen que Cosmos DB sea un servicio muy interesante está que es multi modelo, es decir, con un mismo servicio podemos utilizar cuatro tipos diferentes de persistencia de datos, en este caso: DocumentDB, MongoDB, Azure Table Storage y Gremlin (grafos).

En este articulo vamos a ver cómo usar Azure Table Storage a través de Cosmos DB Table API. Debemos tener en cuenta que esta API se encuentra todavía en *preview*.

**¿Cuándo utilizar Cosmos DB Table API y por qué?**

Cosmos DB Table API proporciona las mismas funcionalidades que Azure Table Storage pero con las ventajas y beneficios del servicio Cosmos DB.

Lo utilizaremos cuando nuestra aplicación requiera las siguientes características:

* Necesidad de guardar los datos con *key-value*.
* Necesidad de tener un esquema flexible (NoSQL).
* Necesidad de geo replicación.
* Necesidad de tener un alto *througput*.

**¿Qué beneficios nos ofrece utilizar Table API?**

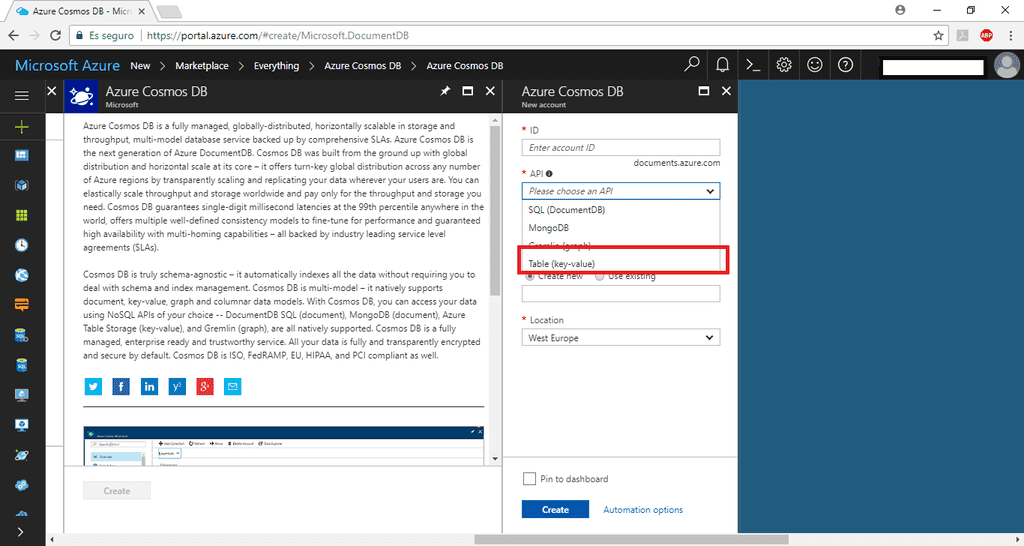
En la siguiente tabla vamos a ver una comparativa ente Azure Table Storage y Cosmos DB Table API.

|  | **Azure Table Storage** | **Azure Cosmos DB: Table storage (preview)** |
| --- | --- | --- |
| **Latencia** | Rápido, pero no tiene límites de latencia. | Para escrituras y lecturas la latencia es de milisegundos a nivel de un solo dígito, siendo la latencia en lecturas menor a 10 ms y menor a 15 ms en escrituras al 99% en cualquier parte geográfica. |
| **Throughput** | Alta escalabilidad, pero no tiene thorugput dedicado por modelo y no por tabla. El límite de escalabilidad es de 20.000 operaciones por segundo. | Alta escalabilidad con throughput dedicado por tabla. No existe límite de throughput y soporta más de 10 millones de operaciones por tabla. |
| **Global Distribution** | Solo permite una sola región y una segunda de lectura para alta disponibilidad. | Permite replicación en más de 30 regiones en cualquier momento en cualquier lugar del mundo. |
| **Indexación** | Solo permite indexar en los índices primarios de la *PartitionKey* y *RowKey,* no en los secundarios. | Indexación automática para todas las propiedades. |
| **Query** | A la hora de ejecutar *querys* se usa el índice de la *primary key.* | Al indexar todas las propiedades, las *querys* son mucho más rápidas, además **Cosmos DB** soporta agregados, *geo-spatial* y ordenación. |
| **Consistencia** | La consistencia es del tipo *strong* para la regiones primarias y *eventual* para las secundarias. | Cinco tipos de consistencias: *Strong, Bounded Staleness, Session, Consistent Prefix y Eventual.* |

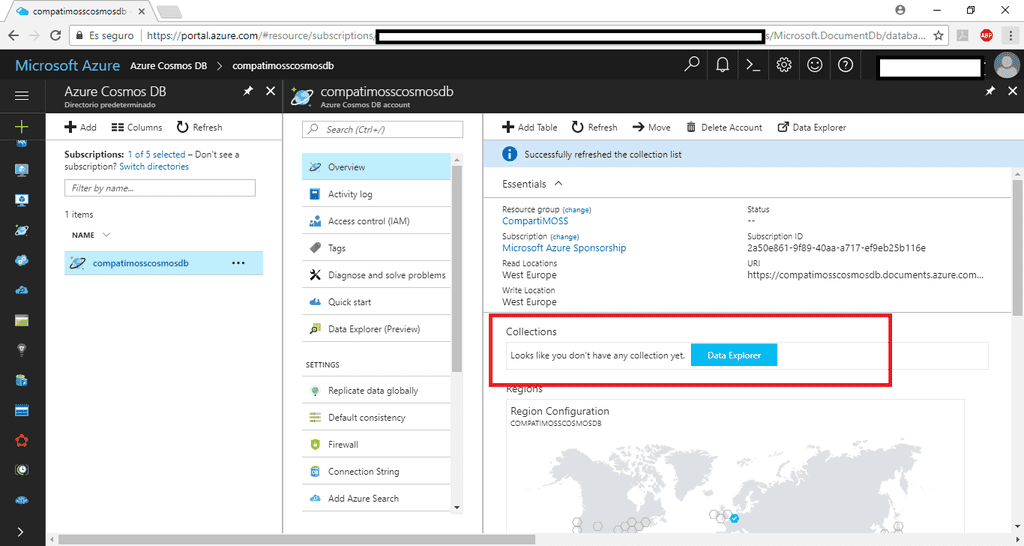
Para más información sobre la consistencia en Cosmos DB consultar: <https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels>

Ahora que ya tenemos claro las ventajas que nos ofrece Cosmos DB Tabe API, vamos a ver cómo utilizarlo. Lo que vamos a ver es un pequeño programa donde vamos a poder comparar las latencias entre Azure Table Storage y Cosmos DB Tabe API. Veremos que para ambos nos sirve el mismo código utilizando la misma librería WindowsAzure.Storage-Premium que está en *preview*.

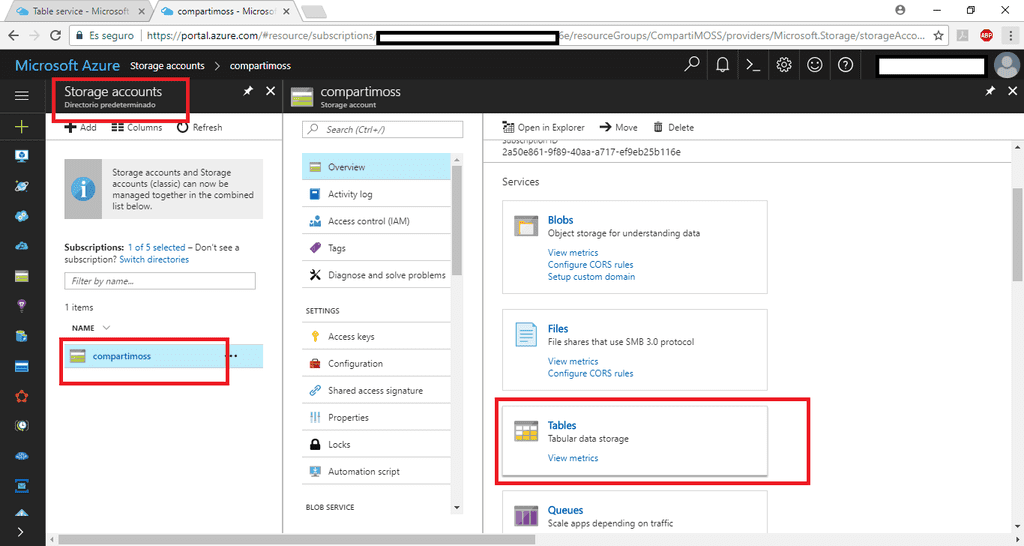
Primero crearemos el servicio CosmosDB en Azure y seleccionaremos Table API

[](https://www.compartimoss.com/static/197e37cfccbce8fd60d9eb7ce006b71f/ad00e/image8.png)

Una vez creada verás que se ha creado el servicio sin ninguna tabla asociada:

[](https://www.compartimoss.com/static/6e11a9a465b840f8eb991a60ab3d72e5/ad00e/image9.png)

También deberéis crear un Azure Storage Account:

[](https://www.compartimoss.com/static/cd469aab55be884f2722ae185ecc218a/ad00e/imagea.png)

Ahora vamos a ver el código, si habéis usado Azure Table Storage veréis que no encontrareis grandes diferencias en las clases y los métodos utilizados. Lo que haremos es realizar operaciones sobre un ​​bucle de 100 elementos, los insertaremos, los obtendremos a partir de su PartitionKey y su RowKey, después realizaremos un update y finalmente los eliminaremos.

Lo primero que haremos es crear la conexión con el servicio:

1CloudStorageAccount storageAccountCosmosDB = CloudStorageAccount.Parse("Your connection String");

2

1CloudTableClient tableClientCosmosDB = storageAccountCosmosDB.CreateCloudTableClient();

2

​Como vemos necesitaremos añadir la *connectionstring* de nuestro servicio, si queremos utilizar Cosmos DB aquí deberemos introducir la conexión de este servicio y si queremos usar Azure Table Storage pondríamos su cadena de conexión.

El cuerpo principal de nuestro programa será el siguiente:

1      public void Run(CloudTableClient tableClient, int numIterations)

2

1      {

2

1          List<ArticlesEntity> items = new List<ArticlesEntity>();

2

1          Console.WriteLine("Creating Table if it doesn't exist...");

2

1          //Create table

2

1          CloudTable table = tableClient.GetTableReference(tableName);

2

1          table.DeleteIfExists();

2

1          CreateTable(table);

2

1

2

1          //Insert Elements

2

1          Console.WriteLine("Insert elements ");

2

1          InsertItemsInTable(numIterations, table, items);

2

1

2

1          //Get Elements

2

1          Console.WriteLine("Get elements: ");

2

1          RetrieveElementsFromTable(numIterations, table, items);

2

1

2

1          //Replace Elements

2

1          Console.WriteLine("Replace elements: ");

2

1          ReplaceElementsInTable(numIterations, table, items);

2

1

2

1          //Delete Elements

2

1          Console.WriteLine("Delete elements: ");

2

1          DeleteElements(numIterations, table, items);

2

1

2

1      }

2

**​¿Cómo insertamos elementos?**

Crearemos un objeto *TableOperation*, que es un objeto que según la inicialización que le hagamos indicará al servicio la operación a ejecutar: insert, delete, etc.

Una vez creado ejecutamos en nuestra conexión creada esta operación, un ejemplo sería:

1TableOperation insertOperation = TableOperation.Insert(item);

2

1tableClientCosmosDB.Execute(insertOperation);

2

​Siguiendo con nuestro programa nuestro método de inserción sería:

1private void InsertItemsInTable(int numIterations, CloudTable table, List<ArticlesEntity> items)

2

1        {

2

1            Stopwatch watch = new Stopwatch();

2

1            List<double> latencies = new List<double>();

2

1            for (int i = 0; i < numIterations; i++)

2

1            {

2

1                watch.Start();

2

1                ArticlesEntity item = new ArticlesEntity()

2

1                {

2

1                    PartitionKey = Guid.NewGuid().ToString(),

2

1                    RowKey = Guid.NewGuid().ToString(),

2

1                    Email = $"{GetRandomString(6)}@mail.com",

2

1                    LastArticleTitle = GetRandomString(50),

2

1                    NumberOfArticles = GetRandomNumber(3)

2

1                };

2

1                // Insert item

2

1                TableOperation insertOperation = TableOperation.Insert(item);

2

1                table.Execute(insertOperation);

2

1                double latencyInMs = watch.Elapsed.TotalMilliseconds;

2

1 Console.Write($"\r\tInsert #{i + 1} completed in

2

1 {latencyInMs} ms.");

2

1                items.Add(item);

2

1                latencies.Add(latencyInMs);

2

1                watch.Reset();

2

1            }

2

1            latencies.Sort();

2

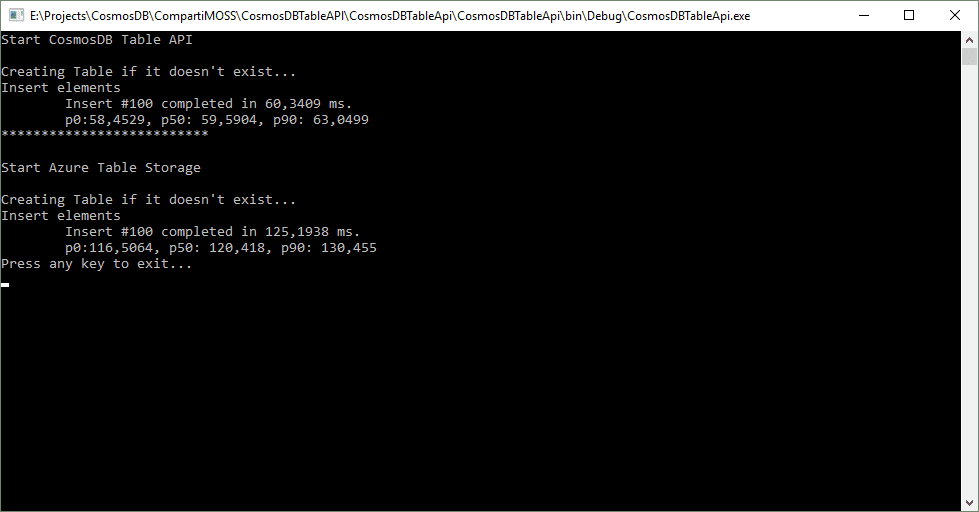
1            Console.WriteLine($"\n\tp0:{latencies[0]}, p50: {latencies[(int)(numIterations \* 0.50)]}, p90: {latencies[(int)(numIterations \* 0.90");

2

1        }

2

Si ejecutamos el programa obtenemos los siguientes resultados:

[](https://www.compartimoss.com/static/78de7c37847545745a097611bbacc0b2/adc48/imageb.png)

**¿Cómo obtenemos elementos?**

Como en el caso de la inserción, crearemos un objeto *TableOperation* mediante su constructor de *Retrieve* y le pasaremos la *PartitionKey* y la *RowKey* a buscar. Ejecutamos en nuestra conexión creada esta operación, un ejemplo sería:

1TableOperation retrieveOperation = TableOperation.Retrieve<ArticlesEntity>(items[i].PartitionKey, items[i].RowKey);

2

1table.Execute(retrieveOperation);

2

​Volviendo a nuestro ejemplo:

1private void RetrieveElementsFromTable(int numIterations, CloudTable table, List<ArticlesEntity> items)

2

1       {

2

1           List<double> latencies = new List<double>();

2

1           Stopwatch watch = new Stopwatch();

2

1           for (int i = 0; i < numIterations; i++)

2

1           {

2

1               watch.Start();

2

1

2

1               TableOperation retrieveOperation = TableOperation.Retrieve<Art

2

1               iclesEntity>(items[i].PartitionKey, items[i].RowKey);

2

1               table.Execute(retrieveOperation);

2

1               double latencyInMs = watch.Elapsed.TotalMilliseconds;

2

1               Console.Write($"\r\tRetrieve #{i + 1} completed in {latencyInMs} ms");

2

1               latencies.Add(latencyInMs);

2

1               watch.Reset();

2

1           }

2

1           latencies.Sort();

2

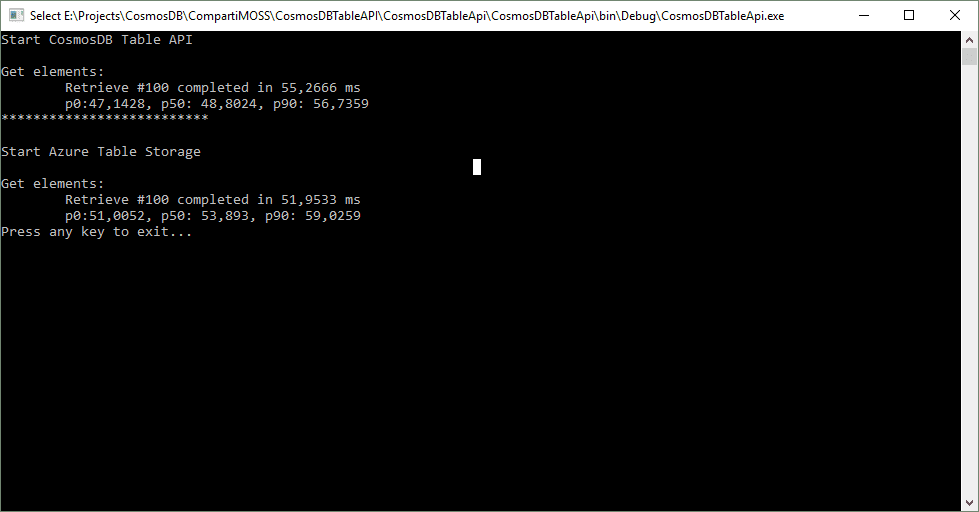
1           Console.WriteLine($"\n\tp0:{latencies[0]}, p50: {latencies[(int)(numIterations \* 0.50)]}, p90: {latencies[(int)(numIterations \* 0.90)]}");

2

1       }

2

​Si ejecutamos obtenemos los siguientes resultados:

[](https://www.compartimoss.com/static/0dc5c9465084a7601740436ed6dbe7eb/adc48/imagec.png)

Podemos observar que al obtener elementos por *Primary Key* la diferencia no es tan grande como en la inserción, pero Cosmos DB sigue siendo más rápido.

**¿Cómo actualizamos elementos?**

Siguiendo los mismos pasos anteriormente comentados. Por ejemplo:

1TableOperation replaceOperation = TableOperation.Replace(items[i]);

2

1table.Execute(replaceOperation);

2

El código de nuestra aplicación:

1      private void ReplaceElementsInTable(int numIterations, CloudTable table, List<ArticlesEntity> items)

2

1       {

2

1           List<double> latencies = new List<double>();

2

1           Stopwatch watch = new Stopwatch();

2

1           for (int i = 0; i < numIterations; i++)

2

1           {

2

1               watch.Start();

2

1               // Same latency as inserts, p99 < 15ms, and p50 < 6ms

2

1               items[i].NumberOfArticles = 4;

2

1               TableOperation replaceOperation = TableOperation.Replace(items[i]);

2

1               table.Execute(replaceOperation);

2

1               double latencyInMs = watch.Elapsed.TotalMilliseconds;

2

1               Console.Write($"\r\tReplace #{i + 1} completed in {latencyInMs} ms");

2

1               latencies.Add(latencyInMs);

2

1               watch.Reset();

2

1           }

2

1           latencies.Sort();

2

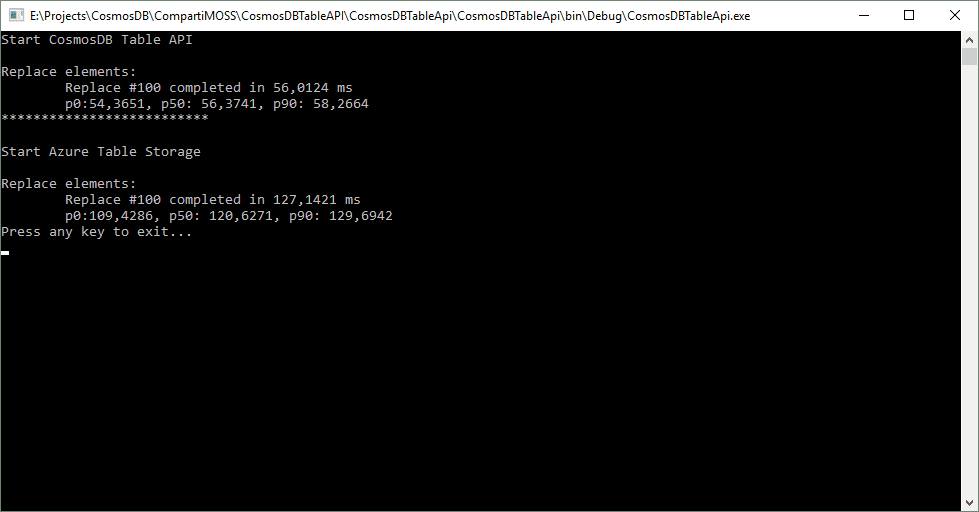
1           Console.WriteLine($"\n\tp0:{latencies[0]}, p50: {latencies[(int)(numIterations \* 0.50)]}, p90: {latencies[(int)(numIterations \* 0.90)]}");

2

1       }

2

​Y si ejecutamos obtenemos:

[](https://www.compartimoss.com/static/0d0c94e6b6aca9fa500b730fd4690538/adc48/imaged.png)

Volvemos a ver que las diferencias se disparan cuando se trata de escritura.

**¿Cómo borramos elementos?**

Directamente ponemos el ejemplo dado que la pauta es la misma que en los casos anteriores:

1TableOperation deleteOperation = TableOperation.Delete(items[i]);

2

1table.Execute(deleteOperation);

2

​El código de nuestra aplicación:

1private void DeleteElements(int numIterations, CloudTable table, List<ArticlesEntity> items)

2

1        {

2

1            List<double> latencies = new List<double>();

2

1            Stopwatch watch = new Stopwatch();

2

1            for (int i = 0; i < numIterations; i++)

2

1            {

2

1                watch.Start();

2

1                // Same latency as inserts, p99 < 15ms, and p50 < 6ms

2

1                TableOperation deleteOperation = TableOperation.Delete(items[i]);

2

1                table.Execute(deleteOperation);

2

1                double latencyInMs = watch.Elapsed.TotalMilliseconds;

2

1                Console.Write($"\r\tDelete #{i + 1} completed in {latencyInMs} ms");

2

1                latencies.Add(latencyInMs);

2

1                watch.Reset();

2

1            }

2

1            latencies.Sort();

2

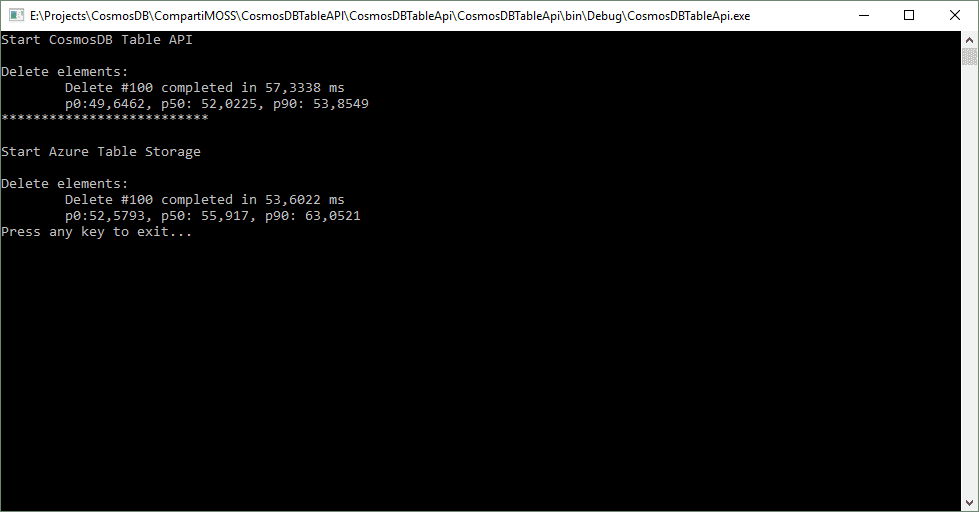
1            Console.WriteLine($"\n\tp0:{latencies[0]}, p50: {latencies[(int)(numIterations \* 0.50)]}, p90: {latencies[(int)(numIterations \* 0.90)]}");

2

1        }

2

​Y el resultado:

[](https://www.compartimoss.com/static/e23f44b9dd65ecc2356d7ddc261cd5ec/adc48/imagee.png)

Vemos que al borrar elementos una vez más las diferencias no son tan grandes, pero sigue siendo mejor Cosmos DB.

**Conclusiones**