Cosmos DB Optimization Workshop

1. To the cloud

**Background story**

Contoso Video currently runs their web application with a backend SQL database on a combination of dedicated and co-located servers through a local, non-global hosting provider. They are concerned about the long-term cost and the time and effort of keeping up with operating system and database patching, backup and restore management time. In the future, they will want to scale the workload geographically.

Contoso Video has had performance and scale issues with the on-premises database. They have provisioned an over-sized server and are hoping it will meet their immediate needs but are worried about future growth. They would like the ability to scale the database instance up and down as needed without the up-front investment in hardware.

Finally, since Contoso is interested in capturing clickstream telemetry from users, they want a database that can handle high-throughput and that will enable them to build an event sourcing pattern to react to incoming data in real-time.

One concern Contoso Video has is how they will replicate data across global instances. Can they enable their web applications that are deployed in the same region as a database instance to write only to that regional instance? If so, how will all those instances resolve data consistency and synchronize without creating conflicts?

**Technical details**

The team has the freedom during the OpenHack to choose the solution they believe will best fit Contoso Video's needs. However, the team must be able to explain the thought process behind the decisions to the team's coach.

**Success criteria**

* Select and provision a database with the following characteristics:
  + Enables a simplified process for scaling up and down
  + Supports a flexible schema with a multi-region, global distribution

**Ramp Up**

* [What Are NoSQL Databases?](https://azure.microsoft.com/overview/nosql-database/)

**Choose Your Tools**

* [Azure Cosmos DB](https://docs.microsoft.com/azure/cosmos-db/introduction)

**TASK TO DO**

1. Setup the environment

<https://github.com/solliancenet/nosql-openhack>

<https://github.com/solliancenet/nosql-openhack/blob/master/deployment-instructions.md>

1. Create a Cosmos DB Database in the resource group
2. Migrate the database to the cloud

**Background story**

Now that the team has provisioned a new database, Contoso Video wants to start migrating their data. They hope that all the data can be migrated without loss, but are not sure how it should be restructured, if at all, to work within the new data store. Contoso informed the team that their website performs satisfactorily in general, but once a customer selects an item to get more detailed information, the page often loads slowly for customers during peak hours. This drives some to become frustrated and to shop on other websites instead.

Contoso is hoping that by migrating their data to the new cloud database, they can independently scale up throughput for product browsing to alleviate these long wait times without affecting the cost or performance of their checkout functionality.

**Technical details**

The team will now begin migrating from the SQL database to the new data store. Migrate all data to the new database to establish a baseline model. This data is primarily queried by category on the main product page, and by specific product when a user selects an item to view. User interactions are stored as they browse the catalog, view items, and perform shopping cart-related actions. The store should be able to support these query patterns with minimal latency. The team can assume that all product categories have a similar number of items, that they are all searched at similar rates, and that new products are introduced infrequently.

In this challenge, the goal is to perform a **raw migration.** The team will optimize their database design in the next challenge. Because of this, the team must ensure they have a repeatable migration process when they complete this challenge.

**Contoso Video resources**

Contoso has provided the team with a copy of their web application and data (with test data only!). These resources are pre-deployed for the team to review a recent version of the web application and SQL Server database. Contoso believes it is important for the team to understand where they are today in order to *plan for the future*. The web application source code is also available for review, if so desired.

**Note**: All resources, including documentation for the existing SQL database can be [found here](https://github.com/julienmichel/azure-cosmosdb-workshop-optimisations).

The team is free to complete the following tasks to review the current application state:

* Open the deployed website, browse the movie catalog, add a movie to the cart.
* Review the database and relational schema.
* Review key queries in the application code.

The SQL database credentials are:

**Username**: cosmosdbworkshopadmin **Password**: Password123

**Success Criteria**

* All data is migrated to the new database. This is a **raw migration** with no expectation of optimization at this stage.
* The team has created and executed queries to count records of each type (Category, Item, Orders, etc.) to validate migration success.
* The team has a repeatable migration process that will allow them to re-migrate the data as needed during the optimization challenge.
* The team must explain to the coach how they would scale this workload up during peak hours.

**Resources**

* [Copy data to or from Azure Cosmos DB (SQL API) by using Azure Data Factory](https://docs.microsoft.com/azure/data-factory/connector-azure-cosmos-db)

**Task to do**

Create a Cosmos DB Database name *Movies* in a new account. Inside create all the container need :

Sample of AZ command to create quickly

* *az cosmosdb sql container create -a {Account Name} -g cosmosdbworkshop -d Movies* *-n Item -p '/ItemId' --throughput 400*
* *az cosmosdb sql container create -a {Account Name} -g cosmosdbworkshop -d Movies* *-n Cartitem -p '/Cartitemid' --throughput 400*
* *az cosmosdb sql container create -a {Account Name} -g cosmosdbworkshop -d Movies* *-n Itemaggregate -p '/id' --throughput 400*
* *az cosmosdb sql container create -a {Account Name} -g cosmosdbworkshop -d Movies* *-n OrderDetails -p '/OrderDetailId' --throughput 400*
* *az cosmosdb sql container create -a {Account Name} -g cosmosdbworkshop -d Movies* *-n Order -p '/OrderId' --throughput 400*
* *az cosmosdb sql container create -a {Account Name} -g cosmosdbworkshop -d Movies* *-n Category -p '/CategoryId' --throughput 400*
* *az cosmosdb sql container create -a {Account Name} -g cosmosdbworkshop -d Movies* *-n User -p '/UserId' --throughput 400*
* *az cosmosdb sql container create -a {Account Name} -g cosmosdbworkshop -d Movies* *-n Event -p '/EventId' --throughput 400*

Create a pipeline to copy data from SQL to Cosmos DB, with no transformation, 1 table = 1 collections .. create all the collections, check the throughput by DB or by collections ..

1. Optimize NoSQL design

**Background story**

Aside from the crucial steps to modernize by moving their data and application to the cloud, at the top of Contoso's mind is whether they can meet the higher demand of exponential growth. This growth will be driven by the movie catalog update with new international content and the audience that goes with it. This growth concern drives them to pursuing optimal performance levels and fueling their desire to unlock new capabilities and even more growth through innovation. They are looking to you as the expert in this area. Yes, they know how their new data store gives them flexibility in their data schema, but what are the fundamental design decisions they need to make within their data layer to stay performant in the future without scaling up to the point of spending more than they should?

**Technical details**

The team will begin evaluating query patterns as stated within the guide and as observed within the sample web application and related artifacts.

At a high level, the following is a set of specific query patterns that Contoso has explained they want to make sure continue to work in the new design, are performant, and are cost-optimized for the following *anticipated* query volumes:

* Retrieve list of top 10 movies by biggest popularity and lowest price *~500 requests/second*
  + SELECT top(10) FROM c order by c.Popularity DESC, c.UnitPrice DESC
* Retrieve the list of categories: *~500 requests/second*
  + SELECT DISTINCT(c.CategoryName) FROM c on category
* Filter movies (items) by category name : *~200 requests/second*
  + SELECT c.CategoryId FROM c where c.CategoryName = "Fantasy"
  + SELECT \* FROM c where c.CategoryId = 14 on category for the details
* Retrieve orders with details showing products with quantities: *~10 requests/second*
  + SELECT \* FROM c where c.OrderId = 50 (ex take here 50 ) on orders
  + SELECT \* FROM c where c.OrderId =50 on orderdetails
  + SELECT \* FROM c where c.ItemId BETWEEN 397519 AND 397524 on item

Contoso currently has 85,000 concurrent users during peak time but would like to plan for 5 million concurrent users.

Perform sizing exercise to determine anticipated scale requirements. Optimize the schema design to reduce scale requirements if possible.

**Success Criteria**

* The team has implemented optimizations and demonstrated to their coach an improvement in query performance, and/or cost per query. *Application modification is not required*.
  + Estimate the cost per query for reads and writes, then use this information to calculate the throughput the design would need at the anticipated scale.
  + Investigate how to scale capacity if needs change hour by hour, then share this plan with the coach.
* The team has created different data models for their database to optimize for the high-level queries above, and to implement denormalization as needed.

**Tips**

* Compare the partition strategy developed in the previous lab to what you think it should be now. Has your plan evolved? Are there any instances where having more than one partition to support different read-heavy workloads make sense?
* When optimizing your schema, consider the following:
  + The optimal partition key has been applied at the collection level for each collection.
  + Related documents/entities are collocated under the same partition. These should be entities that are most often retrieved together.
  + If appropriate, some related data is embedded. For example, if you return a product and want to include some information about the category (which may be stored in a different partition) without retrieving the entire category document. In other words, denormalization.
  + As needed, virtual partition key values are used.
  + Adjust indexing appropriate for read or write-optimized workloads.
  + The most appropriate consistency model is selected based on usage patterns within the web application, and through testing.
* Take care to consider the differences in modeling data for different data stores.
  + Avoid treating collections like tables and understand how multiple entity types can possibly be stored within the same collection.
* [Choosing a Data Model](https://docs.microsoft.com/azure/architecture/data-guide/big-data/non-relational-data)
* [Data Modeling for NoSQL](https://docs.microsoft.com/azure/cosmos-db/modeling-data)
* [Partitioning in Azure Cosmos DB](https://docs.microsoft.com/azure/cosmos-db/partitioning-overview)
* [Modeling and partitioning a real-world example on Azure Cosmos DB](https://docs.microsoft.com/azure/cosmos-db/how-to-model-partition-example)

**Task to do**

* Execute all the query with movieraw
  + Some queries need more than one query, you need to execute the query though portal and calculate the RU
* Make a new model of data with order, ordersdetails and item. Example here :

<https://docs.microsoft.com/en-us/azure/cosmos-db/migrate-relational-to-cosmos-db-sql-api>

* Define optimization
  + Check the need of new model ?
  + Check the need of composite indexes ?

**Queries that can be used as sources in your ADF pipelines** **:**

SQL Query for orders, orders details :

SELECT

id = o.OrderID,

o.OrderDate,

o.FirstName,

o.LastName,

o.Address,

o.City,

o.State,

o.PostalCode,

o.Country,

o.Phone,

o.Email,

o.ReceiptUrl,

o.Total,

o.HasBeenShipped,

(SELECT i.ProductName, i.Description, i.Popularity, i.OriginalLanguage, i.ReleaseDate, od.UnitPrice, od.Quantity FROM OrderDetails od

JOIN Item i ON i.ItemId = od.ProductId

WHERE od.OrderId = o.OrderId FOR JSON PATH) AS OrderDetails

FROM Orders o FOR JSON PATH)

SQL Query for category, item :

SELECT o.\* , i.categoryName FROM Item AS o

        JOIN Category i ON i.CategoryId = o.CategoryId