This blog is not intended to cover CosmosDb in detail, we will glance at some of its main features, then present a use case of integrating it into an existing SQL application. If the reader is already familiar with Cosmos you may skip the intro section.

# Introduction

Document databases gained huge popularity in the past decade. Their flexible and semi structured nature makes it easier for developers to persist and query data, when compared to traditional relational engines. Their hierarchical format facilitates binding database fields to object-model or user interface attributes, especially when dealing with complex entities with multiple collections and nested attributes.

Microsoft CosmosDb is one of the main engines in this space, along with MongoDb Atlas, Google Firestore and Amazon DocumentDB.

CosmosDB also offers distinct features:

* Built-in fault tolerance, making it highly available with 99.999% SLA.
* Low latency and elastic scale. The more your database grows the more compute and storage nodes are leveraged, maintaining the throughput for writes and reads below 10ms for 99% of requests (exceptions apply depending on consistency and distribution).
* Globally distributed. Your database can be spread across regions or continents, you can write anywhere, and the Cosmosdb will sync the changes, without worrying about CDC, replication, log shipping or jobs. It also has built-in proximity features, so if your application is distributed it will automatically communicate with the closest replica.

These features are highly desirable for applications with high throughput or 24/7 uptime, and are difficult to achieve with classic relational engines. Other interesting Cosmosdb features include:

* It’s a Paas service, so you don’t need to worry about patching, backups, or hardware failures.
* Multi-Model APIs, making it easy to migrate from other engines such as Gremlin, MongoDb, Cassandra and Azure Table.
* Flexible billing model. You may allocate resource units to at the database or collection level, or you may opt for serverless and pay for what you use. Many factors play a role in your monthly Cosmos bill (such as load, redundancy, consistency level and whether the analytical store is on), but the most important factors to keep cost low are:
  + For collections with high volume of reads, partition by an attribute frequently used in searches, this leads to the search getting executed only on the node which contains the partition instead of all nodes.
  + For collections with high volume of writes, turn off automatic indexing for the columns not used in searches.
  + Watch your document sizes. Larger documents cost more to read and write. Consider moving your low frequency/large attributes to separate collections.

Document databases also have downsides:

* Schema is evaluated during reads. This is what makes it popular with application developers, but the lack of standardization can lead to poor data quality.
* Lack of constraints for uniqueness or referential integrity.
* Limited support for joins and aggregates. You do have some functionality, but it will be slow and costly compared to relational engines.

Migrating applications from SQL to Cosmos is challenging because the engines have different paradigms. The rise of service-oriented architectures increased data independence across applications. Instead of writing your PC, Web, Mobile and Integration apps all pointing to the same database, many companies have been opting for splitting (or duplicating) data across different database engines, allowing for greater scalability, and leveraging the best features of each engine. A common trend is to keep your transactional workload in a SQL backend, and analytical workload in a Distributed SQL or Spark environment, as an example Synapse covers both use cases, and the latest versions of SQL offer a link to replicate data into Synapse Dedicated Pools. Many companies also have been using document dbs (instead of SQL) for their transactional systems. To improve reportability of your data, Cosmosdb offers the option to turn on the analytical store and link to Synapse. The analytical store flattens your data, converting nested json attributes into a tabular format. The Synapse Link replicates your data into ADLS blob storage in parquet format, allowing you to query with Spark or Serverless SQL Pools. The latency is usually a few seconds.

# CosmosDb as a High-Performance Cache

Our use case came from a project with a major health insurance provider in the United States. They used a SQL backend for their benefits, claims, web portal and customer service systems. They then created a mobile app over the same SQL backend, which increased their traffic significantly, also increasing latency and concurrency issues with SQL. Another challenge was the fact that their database was highly normalized, and when a customer opened their mobile app, they pulled data from several tables. They tried scaling SQL up, enabling read replicas, tuning indexes, but as an SMP engine SQL can only scale so far. They then considered creating a separate database for their mobile app and wanted to store data in a way that facilitated pulling all information about a customer, as a “high performance cache”. This concept has long been implemented with Redis. However, Redis have some limitations the customer was concerned about:

* Redis max db size is 4 GB.
* Redis is essentially an in-memory key value pair store, while you can store json as string you will need more code to serialize/deserialize, and you won’t have built in search functionality in the json attributes.
* Redis is not fault tolerant and does not offer elastic scale, so the app would be prone to outages and slowdowns.

We then opted to use CosmosDb as the cache database. The document model was ideal to store all information about a customer in a few collections and allowed for consistently fast page loads without any impact on SQL. The automatic indexing feature allowed for fast searches. Essentially all the read operations interacted with CosmosDB. The write operations (which were rare) invoked the original APIs over the SQL backend and triggered our custom “SQL To Cosmos Replication”. During the first phase all changes were still made in SQL. When users called their customer service to make a change, or changed their coverage during open enrollment, the updates would be done to SQL, leveraging mature routines and constraints, and our custom replication would update Cosmos within a few seconds. This architecture allowed to remove indexes in SQL and to eliminate read replicas, reducing the overall cost while improving end user experience.

# Cosmos To Sql Replication

This is an example of how to track changes in SQL and replicate to Cosmos. Our sample code should be used as a reference so the reader can understand the concept, then implement and test their own version.

Here is the repo ink: <https://github.com/MarcoABCardoso/sql-cosmos-replication>

It is composed of 2 parts:

1. The SQL script creates the following objects under cosmos schema:

|  |  |  |
| --- | --- | --- |
| Type | Name | Purpuse |
| Table | ImportLog | Stores history of execution. Each run uses “high watermark” method with range DateFrom and DateTo values, both as datetime.  The DateFrom takes the last successful DateTo, and DateTo gets the current system time.  Other columns track execution time and how many documents were processed. |
| Table | Collections | Defines which collections in Cosmos need to be updated, each collection is implemented by a custom stored procedure. |
| Procedure | spStart | To start the sync process |
| Procedure | spEnd | To finalize the sync process |
| Procedure | spGetPersonsToRefresh | This procedure gets the Persons that changed since last execution.  Change tracking is based on column [ModifiedDate], and we are tracking tables Person, PersonPhone and EmailAddress.  We assume that no hard deletes are executed, and that all processed that modify Person update the ModifiedDate column. |
| Procedure | spGetPersonJSON | This proc converts a person record to json, including their phones and emails. We leverage the FOR JSON native function. |
| Function | fnGetPersonJSON | This function does the same thing as the prior procedure, but allows for inline execution. |
| Procedure | spGetPersonsToRefreshV2 | This function returns all the persons which changed within a date range, and their JSONs.  This proc is redundant but serves to illustrate the use in multiple ways.  Notice the execution is very fast. In our dev environment we got 10k person jsons in 1 second. |

Our example uses the AdventureWorks2019 on SQL 2019 Developer, which can be downloaded on the links below.   
Our sample should work in any supported version of SQL, including Azure SQL and Managed Instance.   
<https://docs.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver16&tabs=ssms>  
[SQL Server Downloads | Microsoft](https://www.microsoft.com/en-us/sql-server/sql-server-downloads)

1. This is the C# console app gets results from spGetPersonsToRefreshv2 and saves to Cosmos using the UpsertItemStreamAsync method.

During the first execution there will be no history, so it will pick up all persons. On later runs it will pick up only the people that had a change.

You can opt to process each person when they get updated using spGetPersonJSON, or to schedule a periodic run to process all persons that changed.