Microsoft

MongoDB to Cosmos DB Migration via Azure Databricks

Prepared by

Data SQL Ninja Engineering Team (datasqlninja@microsoft.com)

Table of Contents

1 Summary

1.1 Scope & Objective

This guide offers architectural guidance on how to migrate and transform complex NoSQL data from MongoDB to Azure Cosmos DB.

1.2 Why do this?

- Azure Cosmos DB offers predictable response times with ability to scale compute, memory and storage far beyond what is possible with MongoDB
- Azure Cosmos DB offers same-or-better performance than MongoDB for many use cases
- Azure Databricks enables rapid development and iteration of transformation code as well as support for transformation of complex data types within JSON documents

1.3 Recommendations

- Use Azure Cosmos DB in SQL API mode for best performance
- Remodel the data for optimal Azure Cosmos DB access patterns
- Use Azure Databricks (Spark) for SQL-like transformations and loading at scale
- Use the Mongo DB Spark Connector for bulk reading of MongoDB into Spark
- Use the Cosmos DB Spark Connector for bulk loading into Azure Cosmos DB

2 Design Considerations

2.1 Point Lookups

Point Lookups are an alternative mechanism for retrieving a document. If you supply the Partition Key and the Row Key value, you can bypass the query optimizer and retrieve the document directly.

The "id" field is a system field in Azure Cosmos DB that is added automatically to every new item, but can also be populated manually, providing certain rules are obeyed.

2.2 SQL API

For optimal performance, use Azure Cosmos DB in SQL API mode. Although Cosmos DB is also feature compatible with MongoDB's Wire Protocol, the SQL API underpins all other APIs available on Cosmos DB, offering the minimum throughput overhead.

2.3 Collocated Items

Storing structurally similar and related items in the same container (partitioned by the same field – UUID) allows the database to retrieve these closely related items with less reads and avoiding 'fan-out', minimizing overall response times.

2.4 Azure Databricks

Apache Spark is an excellent choice for migrating and transforming complex data types.

Databricks provides an easy to configure 'single view' of the Spark cluster, taking care of all the background processes around starting new nodes etc.

In a feature-rich notebook environment, we can create Scala, Python & SQL code to load and manipulate large data sets very quickly thanks to the in-memory distributed data processing capability of Spark.

2.5 Bulk Executor for Cosmos DB

Spark is also useful for migrating large data sets because of the Spark Connector for Cosmos DB, which makes use of the Bulk Executor library for Cosmos DB. The Bulk Executor requires the SQL API, which also feeds into the decision to go with SQL API rather than MongoDB API for the target.

Key features of the bulk executor library

- It significantly reduces the client-side compute resources needed to saturate the throughput allocated to a container. A single threaded application that writes data using the bulk import API achieves 10 times greater write throughput when compared to a multi-threaded application that writes data in parallel while saturating the client machine's CPU.
- It abstracts away the tedious tasks of writing application logic to handle rate limiting of request, request timeouts, and other transient exceptions by efficiently handling them within the library.
- It provides a simplified mechanism for applications performing bulk operations to scale out. A single bulk executor instance running on an Azure VM can consume greater than 500K RU/s and you can achieve a higher throughput rate by adding additional instances on individual client VMs.
- It can bulk import more than a terabyte of data within an hour by using a scale-out architecture.
- It can bulk update existing data in Azure Cosmos DB containers as patches.

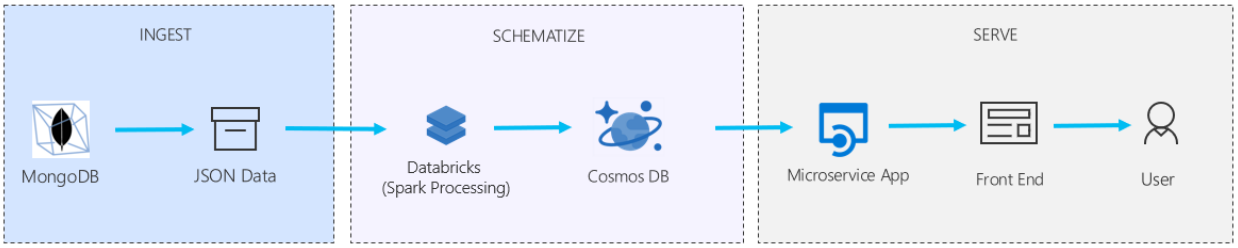
> ⓘ Note
>
> Currently, bulk executor library supports import and update operations and this library is supported by Azure Cosmos DB SQL API and Gremlin API accounts only.

2.6 Extract / Transform / Load

The ETL approach uses a Scala notebook in Azure Databricks. Although Databricks notebooks also support Python and SQL, Scala may be considered for the following reasons;

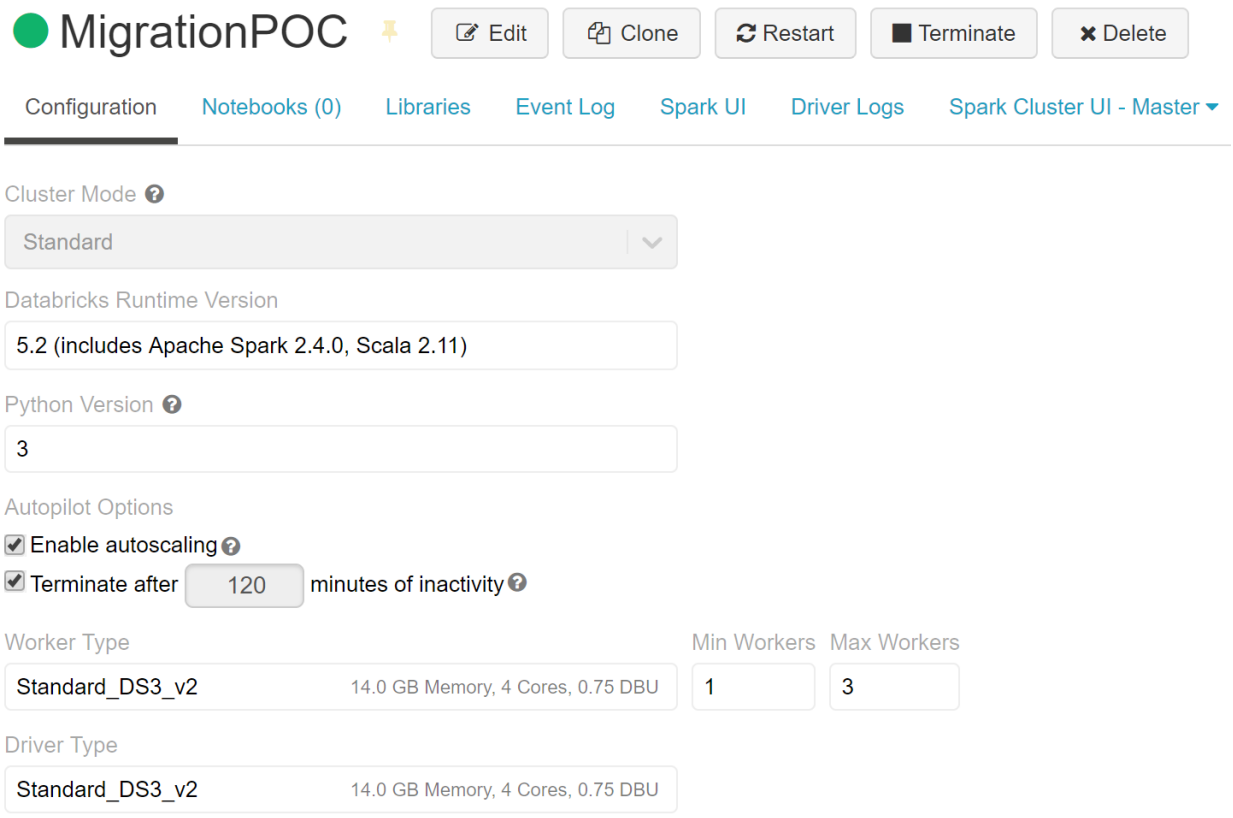
- Scala has been reported in the community as performing up to ten times faster than Python for Spark workloads
- Spark was developed in Scala – it's the native language of the cluster
- Scala was designed to make use of Java libraries, which enables rapid prototyping – most data transformation problems have already been solved in Java, this helps to avoid re-inventing the wheel

3 Migration Architecture



3.1 Setup Guide

3.1.1 Deploy Azure Databricks Spark Cluster



3.1.2 Install the MongoDB Connector for Spark

Add the Mongo connector library to the spark cluster via the Maven coordinates;

`org.mongodb.spark:mongo-spark-connector_2.11:2.3.1`

Figure out the URI for MongoDB, based on its IP address.

```
mongodb://<private ip of mongodb>:27017/admin
```

In the Databricks cluster configuration, click the Spark configuration tab, edit the cluster config

Assemble and enter the Mongo DB connection details into the spark configuration field;

```
spark.mongodb.output.uri mongodb://<private ip of mongodb>:27017/admin
spark.mongodb.input.uri mongodb:// <private ip of mongodb>:27017/admin
```

Hit Save & then restart the cluster.

We should now be able to connect to MongoDB from Databricks

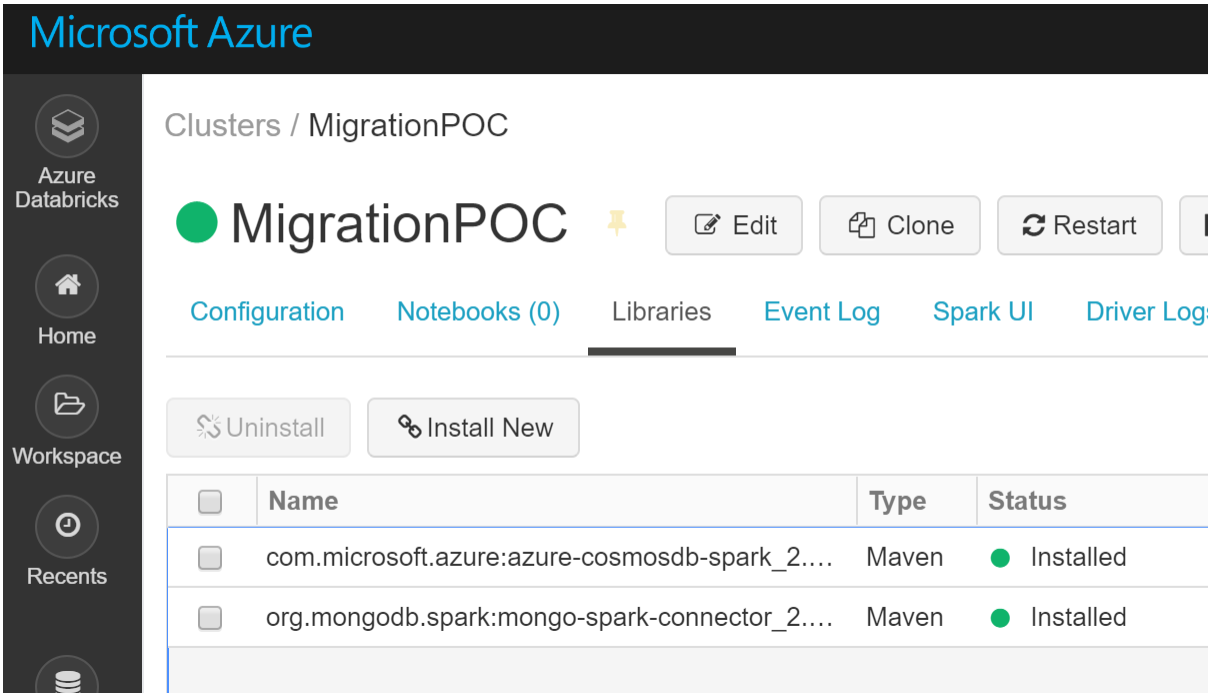
3.1.3 Install the Azure Cosmos DB Connector for Apache Spark

Deploy the latest Cosmos DB connector library to the spark cluster via its maven coordinates;

```
com.microsoft.azure:azure-cosmosdb-spark_2.4.0_2.11:1.4.0
```

3.1.4 Example: Installed Libraries

When both libraries have been installed correctly, it will look something like this;



4 Migration

4.1 Developing your custom migration process

Once the architecture is in-place, it's time to start putting together a Databricks notebook that will extract the data from MongoDB into a Spark Data Frame, then transform the Data Frame, before loading it into a Cosmos DB container.

A Databricks notebook can drive this entire process, end-to-end. Some coding is required, but in return you get a powerful and completely customizable migration workflow, that can handle complex data types, such as nested structs and arrays in JSON documents.

4.2 Example Databricks Notebook

This notebook uses Scala examples to import an entire collection of documents from MongoDB, display them, and write them to Azure Cosmos DB (SQL API).

Requirements;

- A MongoDB server
- Azure Cosmos DB (SQL API)
- Mongo Spark Connector
- Azure Cosmos DB Spark Connector

```
// Use the Spark connector for MongoDB to read the whole collection into a
Spark Dataframe

import com.mongodb.spark._

val UserMongo =
spark.read.format("com.mongodb.spark.sql.DefaultSource").option("database",
"test").option("collection", "user").load()

// List out all the field names in the MongoDB data. We'll be splitting this
data into 3 new collections; Users, Followers, Following.

UserMongo.columns;

// Create a new DataFrame called "User" that doesn't contain the "Followers"
or "Following" fields.

// * Manually populating "id" with "UUID" to enable Azure Cosmos DB point
lookups by UUID
```

```
val User =
UserMongo.select("UUID","_class","_id","birthdate",”realname","salt","state",
"subscriptions","sync_time","tz","userkey","username","website")
.withColumn("id", $"UUID")
// Check count of items in the data frame
User.count()
// Display a maximum of 1,000 items in an interactive window
display(User)
// Create three new DataFrames for "UserAliasEmail", "UserAliasUsername", and
"UserAliasUserKey"
import org.apache.spark.sql.functions._
import org.apache.spark.sql.functions.{concat, lit}
import scala.util.matching.Regex
import java.util.Base64
import java.nio.charset.StandardCharsets

// User defined function to encode the "id" field as Base64 by UTF8
def Base64UTF8 = udf((s: String) => {
  val encoded =
Base64.getEncoder.encodeToString(s.getBytes(StandardCharsets.UTF_8))
  encoded
})

// each item is subjected to the following order of data cleansing to permit
loading into the "id" (rowkey) field of Cosmos DB
// 1 - Truncate field to 100 characters
// 2 - Convert to lower case
// 3 - Convert to Base64 via UTF8 byte array
// 4 - Replace "/" with "!" (Slash is ignored by Base64 but would not be
permitted by the Cosmos "id" field)
// 5 - Concatenate with "AliasType" to differentiate lookups within same
container

val UserAliasEmail = UserMongo.select($"email" as "Alias",
  lit("Email") as "AliasType",
```

```
  concat(lit("Email;"),regexp_replace(Base64UTF8(lower(substring($"email", 1,
100))),"/","!")) as "id",$"UUID")

val UserAliasUsername = UserMongo.select($"username" as "Alias",
  lit("Username") as "AliasType",

concat(lit("Username;"),regexp_replace(Base64UTF8(lower(substring($"username"
, 1, 100))),"/","!")) as "id",$"UUID")

val UserAliasUserKey = UserMongo.select($"userkey" as "Alias",
  lit("UserKey") as "AliasType",
  concat(lit("UserKey;"),$"userkey") as "id",$"UUID")

display(UserAliasEmail)
display(UserAliasUsername)
display(UserAliasUserKey)

// Create a new DataFrame for "Following"
import org.apache.spark.sql.functions._
import org.apache.spark.sql.functions.{concat, lit}

// explode the items into new items with 1000 array elements each
val n = 1000
var f1 = UserMongo.select($"UUID" as
"UUID_F",$"Following",posexplode($"Following.username") as
Seq("pos","uname"), ceil(size($"Following")/n).as("groups"))

// join the individual rows to the users table
val Following = f1.join(User, f1("uname") === User("username"), "left_outer")
.select($"UUID_F" as "UUID",$"username",$"pos",$"UUID" as "Following")
.withColumn("id",  concat(lit("Following;"),floor(col("pos")/n)+1)) // needs
to be a string or spark connector wont load it to cosmos
.groupBy("UUID", "id")
//.agg(collect_list($"Following").alias("Following")) // show UUID without
label
```

```
//.agg(collect_list(array($"Following",$"username")) as "Following") // show username & UUID without labels
.agg(collect_list(struct($"Following" as "UUID",$"username" as "Username")) as "Following") // show username & UUID with labels
.sort("UUID","id")
display(Following)
Following.count()
// Create a new DataFrame for "Followers"
import org.apache.spark.sql.functions._
import org.apache.spark.sql.functions.{concat, lit}

// explode the items into new ones with 1000 array elements each
val n = 1000
var f1 = UserMongo.select($"UUID" as "UUID_F",$"Followers",posexplode($"Followers.username") as Seq("pos","uname"), ceil(size($"Followers")/n).as("groups"))

// join the individual rows to the users table
val Followers = f1.join(User, f1("uname") === User("username"), "left_outer")
.select($"UUID_F" as "UUID",$"username",$"pos",$"UUID" as "Followers")
.withColumn("id",  concat(lit("Followers;"),floor(col("pos")/n)+1)) // needs to be a string or spark connector wont load it to cosmos
.groupBy("UUID", "id")
//.agg(collect_list($"Followers").alias("Followers")) // show UUID without label
//.agg(collect_list(array($"Followers",$"username")) as "Followers") // show username & UUID without labels
.agg(collect_list(struct($"Followers" as "UUID",$"username" as "Username")) as "Followers") // show username & UUID with labels
.sort("UUID","id")
display(Followers)
// Configure write connection to Azure Cosmos DB 'user' collection
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.config.Config
```

```
val WriteConfigUser = Config(Map(
  "Endpoint" -> "https://ascosmostest.documents.azure.com:443/",
  "Masterkey" -> "<removed>",
  "Database" -> "test",
  "Collection" -> "user",
  "Upsert" -> "false"
))

// Save the Users dataframe to Cosmos DB
import org.apache.spark.sql.SaveMode
User.write.mode(SaveMode.Overwrite).cosmosDB(WriteConfigUser)
// Configure write connection to Azure Cosmos DB 'useralias' collection
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.config.Config

val WriteConfigUserAlias = Config(Map(
  "Endpoint" -> "https://ascosmostest.documents.azure.com:443/",
  "Masterkey" -> "<removed>",
  "Database" -> "test",
  "Collection" -> "useralias",
  "Upsert" -> "false"
))

// Save the UserAliasEmail dataframe to Cosmos DB
import org.apache.spark.sql.SaveMode
UserAliasEmail.write.mode(SaveMode.Overwrite).cosmosDB(WriteConfigUserAlias)

// Save the UserAliasUsername dataframe to Cosmos DB
import org.apache.spark.sql.SaveMode
UserAliasUsername.write.mode(SaveMode.Overwrite).cosmosDB(WriteConfigUserAlia
s)
```

```
// Save the UserAliasUserKey dataframe to Cosmos DB
import org.apache.spark.sql.SaveMode
UserAliasUserKey.write.mode(SaveMode.Overwrite).cosmosDB(WriteConfigUserAlias
)
// Configure write connection to Azure Cosmos DB 'social' collection
import com.microsoft.azure.cosmosdb.spark.schema._
import com.microsoft.azure.cosmosdb.spark._
import com.microsoft.azure.cosmosdb.spark.config.Config

val WriteConfigSocial = Config(Map(
  "Endpoint" -> "https://ascosmostest.documents.azure.com:443/",
  "Masterkey" -> "<removed>",
  "Database" -> "test",
  "Collection" -> "social",
  "Upsert" -> "false"
))

// Save the Following dataframe to Cosmos DB
import org.apache.spark.sql.SaveMode
Following.write.mode(SaveMode.Overwrite).cosmosDB(WriteConfigSocial)

// Save the Followers dataframe to Cosmos DB
import org.apache.spark.sql.SaveMode
Followers.write.mode(SaveMode.Overwrite).cosmosDB(WriteConfigSocial)
```

5 Observations

5.1 Load Performance

During tests, an Azure Cosmos DB SQL API database was configured with 50,000 RUs, and connected to a three node Azure Databricks cluster (Standard_DS3_v2).

Throughput was defined at the Container level to prevent contention of resources.

A full load of ~27 million documents completed in 40 minutes. Although the throughput rate was exceeded occasionally, Spark will retry four times by default, so the load was successful overall.

When planning for production migration, use this information to experiment with scale-out of Spark node count and Cosmos DB Request Unit allocation to find your best ratio for optimal throughput.

5.2 Cost Management

The fastest way to remove all items in a collection is to delete the container and recreate it. During development of this migration architecture, it may be necessary to destroy and recreate your containers a few times to support testing. Each time a new container is created, the subscription will be billed for the next hour of provisioned Request Units.

This can result in brief periods of time where overlapping billing for multiple instances of 'one hour' will occur. Although the overall cost impact is low, it is advisable to test data structure locally in Azure Databricks prior to loading into Azure Cosmos DB where possible.

For ongoing use of Azure Cosmos DB, regular recreation of containers should be avoided.

6 References

Azure Cosmos DB bulk executor library overview

https://docs.microsoft.com/en-us/azure/cosmos-db/bulk-executor-overview

Azure Cosmos DB Connector for Apache Spark

https://github.com/Azure/azure-cosmosdb-spark/wiki

MongoDB Connector for Spark

https://docs.mongodb.com/spark-connector/master/