Mirroring in Fabric - MongoDB

Overview

Mirroring in Fabric allows data in the target OneLake tables to be kept in sync with the data in the original source, MongoDB Atlas collection in this case. With mirroring, you can ensure that the latest data is available in OneLake and run multiple forms of analytics on top of it like Spark notebooks, SQL queries, KQL queries or even visualize data using Power BI reports. This is powerful as it brings real time analytics, AI and BI on unified data gathered from different silos.

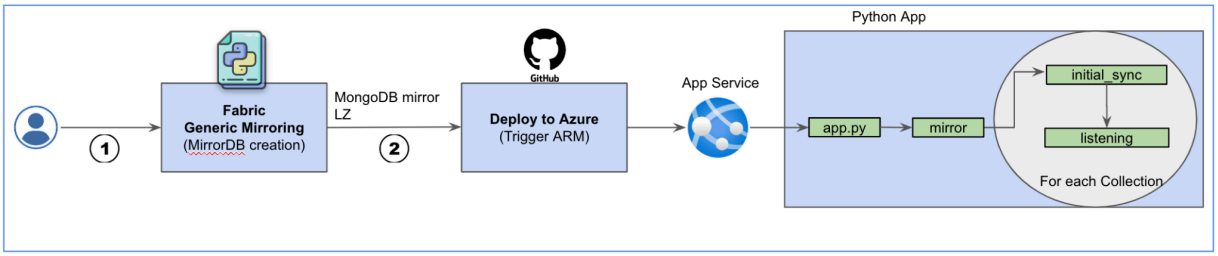
Today, there are multiple ways data from MongoDB Atlas can be brought into OneLake in Fabric. “Data Pipeline” and “Dataflow Gen2”, both are easy mechanisms to bring data from MongoDB Atlas to OneLake in Batches or micro-batches. Mirroring, however, once started will ensure the data is kept in sync between MongoDB Atlas and OneLake, thus enabling real-time analytics, AI based predictions and BI reports.

Goals

This implementation of mirroring will benefit customers to mirror data from MongoDB Atlas to OneLake leveraging the open mirroring capabilities. The Mirrored DB feature in Fabric UI can be used to trigger and set up mirroring DB and Landing Zone for MongoDB Atlas. Once the landing zone is created, a click of a button in Github repo will create an App service, deploy the python app and execute the scripts to trigger Initial Sync and Real time sync.

Architecture

As a prerequisite to the mirroring, create a MongoDB mirror database and provide the user with the Landing Zone (LZ) url. The LZ url will be required to trigger the script for mirroring. The LZ is created as part of the Fabric experience with Mirrored database(preview) feature to create the MirrorDB and the LZ within it as explained in the section here.



The "***Deploy to Azure "*** button when clicked will invoke the ARM template that will take to the App service config screen in Azure. The configuration parameters are added to the App service as environment variables. Once all App service related and the environment variables are filled and the ***"Create"*** button is clicked, the App Service will get deployed with the environment variables populated and the python app deployed and started. The python app basically consists two scripts which are triggered one after other for each collection:

- Real time sync leverages change streams by listening to them , detecting changes and writing to a parquet file using a row marker to identify the change event.
- Python script to do initial sync by reading from MongoDB collection, creating a parquet file and pushing the file into OneLake.

Details of both the scripts are given in the Technical explanation section below.

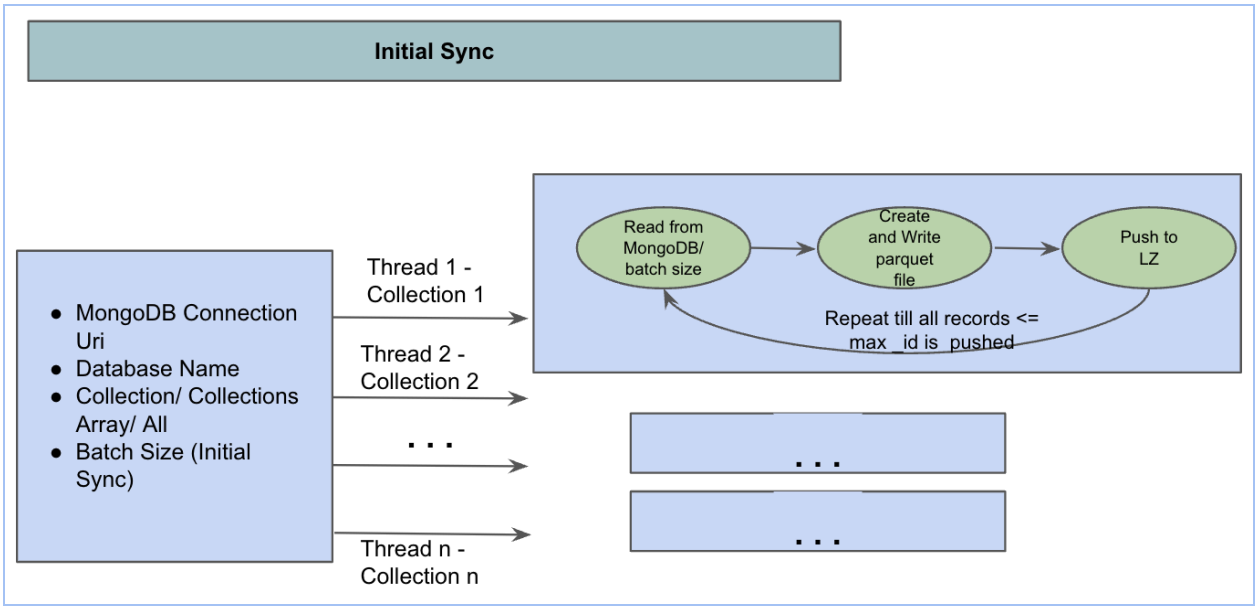
Technical Explanation

I. MirrorDB Creation

To create the mount for replication, now we can use the Fabric UI itself.

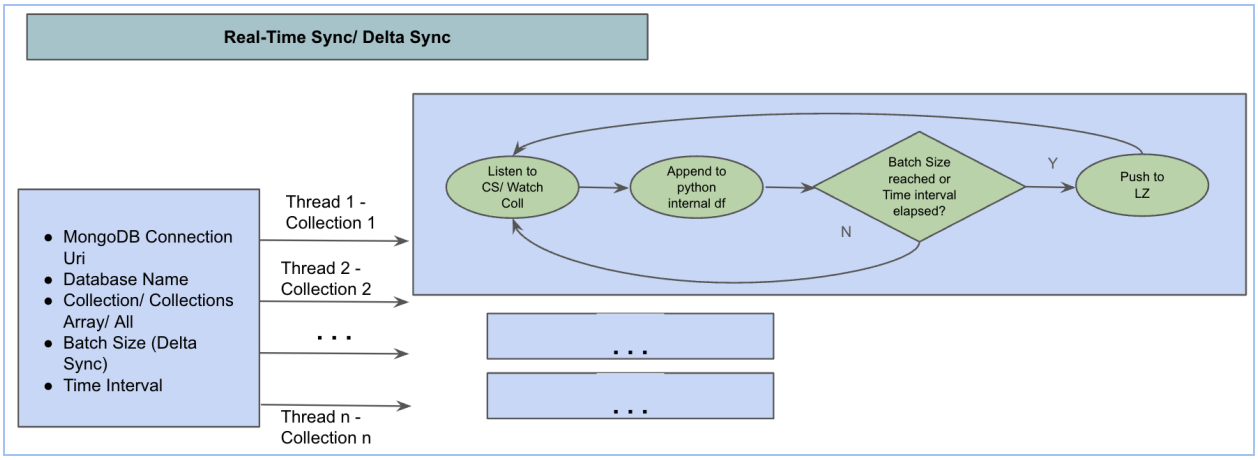
The steps are detailed in the Github link under Step1.

II. Initial Sync



This script is initiated when the mirroring is triggered and starts loading the historical data in MongoDB database to OneLake. The environment variables are required before starting the script. The environment variables will capture the MongoDB connection uri, the database, the collections and the batch size to split the entire process into multiple batches etc. The Initial Sync has the option to take one collection, an array of collections or all collections for the given database. If the user specified an array of collections or all collections, then a thread is started for all the collections in the database. For each thread (i.e for every collection), the entire process is split into batches depending on the specified batch size. For e.g, if the batch size is 10000, then the Initial Sync will read 10000 records from MongoDB, write the records into a parquet file and push that parquet file into OneLake. This process repeats till all existing records are moved to OneLake.

III. Real time sync (RTS) / Delta Sync



The real time sync is also triggered along with the Initial Sync to capture any real time changes occurring during the Initial Sync. The real time sync is a listener python script that listens to the MongoDB changestream for the collections specified by the user and writes the change event with a Rowmarker onto a parquet file and pushes it to LZ. Again there is a Batch size that needs to be specified so that the number of records specified in the Batch size is reached in a parquet file, it is pushed to LZ and a new parquet file is started for the next set of changes.

There are some differences in the logic for the RTS during the Initial sync vs the RTS after the initial sync. During the initial sync, the RTS will listen to changes and capture them and write to parquet files as per the Batch size but just not push them to LZ. Also, for inserts during the initial sync the parquet file will have a Row marker of Upsert, so that if the row is also added by Initial sync, it's just updated and not inserted again. Updates and Deletes will be kept with the same Row marker as updating twice and not finding a record for deletion will not cause any functional issues.

Once the initial sync is over, the accumulated parquet files of RTS during initial sync are renamed with a new sequence number which is obtained by adding 1 to the last Initial sync file's sequence number and pushed to the LZ. Once the cached RTS parquet files are pushed to LZ, the RTS resumes and now onwards when the Batch size is reached, it pushes the parquet file to LZ and starts capturing the RTS in a new file.

Note that if real time updates are required, then Batch Size for RTS should be kept as 1, so that every record is pushed to LZ as and when received and not accumulated in the parquet file. Otherwise, RTS will accumulate the records and when the RTS Batch size is reached, write that parquet file to LZ. There is also a Threshold time parameter which is checked whenever a new event in changestream comes and it checks if the time since last file was written into LZ is more than the Threshold time, in which case it will push the file to LZ even if the Batch size is not reached. The caveat is that the time gets checked only when an event comes in the changestream, else even if the Time has exceeded the file cannot get pushed. It will get pushed when the next event in changestream arrives.

IV. Restart Logic:

A. Initial Sync

Whenever Initial Sync starts, it writes a "init_sync_status" file with a value of "N" into the LZ. So, if this file exists, it means that the script failed before completing initial sync to LZ. Thus, Initial sync will resume from the last _id from the "last_id.pkl" file. It will also get the last parquet file number from the "last_created_parquet.pkl" file and always write a parquet file with the next incremental number. Whenever initial sync completes, it updates the "init_sync_status" file with a value of "Y" which means Initial Sync has completed and thus it will skip Initial Sync and start with RTS only.

B. RTS/ Delta Sync

In the case of RTS, changes are added to a pandas dataframe as and when they arrive. Once the RTS Batch size is reached, they are written to LZ adding 1 to the last parquet_file number. The resume token of the last record is also updated in the “resume_token.pkl“ file.

Whenever a RTS starts, it always checks if the resume_token.pkl file exists. If it exists it will always start listening to the changestream from the resume_token onwards which means any changes that occurred post the last RTS change that was written into LZ, will all be copied in the correct sequence.

V. Real Time changes during Initial sync:

In the case of RTS, changes are added to a pandas dataframe as and when they arrive. Once the RTS Batch size is reached, it will check if the initial_sync is complete or not by checking the value of “Y” in the “init_sync_status“ file, indicating completion of initial_sync. If initial_sync is not “Y”, thus not complete, it will write a parquet file with a TEMP prefix. Whenever another change comes and notices that initial_sync is finished, it will first push all the TEMP parquet files and then write this new event into the dataframe.

They are pushed to LZ, when the initial Sync is completed by renaming them without TEMP and getting a number one greater than the last parquet file pushed to LZ. Note that delete and updates are sent with the same rowmarker as deleting a record if it doesn't exist and updating the record again will not cause any issues. But if its a record Inserted, the row marker is sent as “Upsert '' so that it will update if row exists else insert.

Post initial sync and flushing of TEMP files, it will directly push the RTS files into the LZ directly from the dataframe whenever the Batch size is reached.

VI. Schema and Data type handling:

As data in Fabric tables is relational, we have to make sure that the parquet files being written follow the same schema. Parquet files have a defined structure, so if we are adding a column for a particular record, the other records should have NULL value for that column. Adding a column or removing in the parquet file will be handled by replication.

Also, we have to take care of the data types. The data type mapping that was followed is detailed in this section. Also, if the data type changed from the previous record to the current one, we have to handle the same as the same column cannot have two different data types. That is also indicated in the same spreadsheet, under the “matrix” tab.

Getting Started

I. Trigger mirroring from MongoDB into Fabric OneLake

Follow the GitHub link here to get started with mirroring for MongoDB.

References

i. Data Types Handling - Schema enforcement

Datatype conversion stages:

MongoDB type	pandas column dtype	pandas item type (python)	parquet type	Fabric type
Int32	int64 -> Int64	numpy.int64	int64	bigint
Int64	int64 -> Int64	numpy.int64	int64	bigint
String	object	str	string	varchar
Date	datetime64[ns] -> datetime64[ms]	pandas.Timestamp	timestamp[ms]	datetime2
Boolean	bool -> boolean	numpy.bool_	bool	bit
Double	float64	numpy.float64	double	float
Null	object	NoneType -> str	string	varchar
Data types currently force converted to String but could be carried over unconverted				
Object	object	dict	struct<key: actual_type>	actual_type
Array	object	list	list<element: actual_type>	ERROR

Datatype conversion handling when a new row comes with a different datatype than previously set schema:

Source↓ | Target→	Int64	String	Date	Boolean	float64	Null
Int64		String	Date (ms time since 1970-01-01)	0->False 1->True (other)-> Null	Double	String
String	Int64/Null		Date (if the string is in some date format) / Null	"0", "false"->False "1", "true"->True (other)-> Null	Int64/Null	String
Date	Null	String		Null	Null	String
Boolean	0/1	String	Null		0/1	String
float64	Int64(rounded)	String	Date (ms time since 1970-01-01)	0->False 1->True (other)-> Null		String
Null	Null	Null	Null	Null	Null	

ii. Ignite artifacts:

Blog-MongoDB (contains a short video also linked in)

Blog-Microsoft