Lab - Design Hybrid transactional and analytical processing using Azure Synapse Link

Duration: 60-90 minutes

The main task for this exercise is as follows:

1. Create an Azure Cosmos DB database
2. Configure Azure Synapse Link with Azure Cosmos DB
3. Query Azure Cosmos DB with Apache Spark for Synapse Analytics
4. Query Azure Cosmos DB with serverless SQL pool for Azure Synapse Analytics

Task 1: Create an Azure Cosmos DB database.

1. In the Azure portal, click on the **Home** hyperlink.
2. Navigate to the **+ Create a resource** icon.
3. In the New screen, click in the **Search the services and marketplace** text box, and type the word **Cosmos**. Click **Azure Cosmos DB** in the list that appears.
4. In the **Azure Cosmos DB** screen, click **Create**.
5. Select the **Cosmos DB for NoSQL** and click **Create**
6. From the **Create Azure Cosmos DB Account for NoSQL** screen, create an Azure Cosmos DB Account with the following settings:
    - In the Project details of the screen, type in the following information
        - **Subscription**: the name of the subscription you are using in this lab
        - **Resource group**: **synapse-xx-rg**, where **xx** are your initials
    - In the Instance details of the screen, type in the following information
        - **Account name**: **cosmosdbxx**, where **xx** are your initials.
        - **Location**: same as your resource group.
        - **Capacity mode**: Select **Provisioned throughput**
        - **Apply Free Tier Discount**: Select **Do No Apply**
        - Uncheck **Limit the total amount of throughput that can be provisioned on this account**
        - Leave the remaining options to the default settings
7. In the **Create Azure Cosmos DB Account for NoSQL** blade, click **Review + create**.
8. After the validation of the **Create Azure Cosmos DB Account** blade, click **Create**.
9. When the provisioning is complete, the "Your deployment is complete" screen appears, click on **Go to resource**.
10. In the Cosmos DB screen, click on the **Overview** link.
11. In the **cosmosdbxx** screen, click on **Azure Synapse Link** under Integrations hub and click on Enable synapse link and wait until its enable.

12. Now click on **Overview Link** in **cosmosdbxx** screen, click **+ New Container**. This opens up the **cosmosdbxx Data Explorer** screen with the **New Container** blade.

13. In the **New Container** blade, create a CustomerProfile database with a container named OnlineUserProfile01 with the following settings:

    - **Database id create new**: **CustomerProfile**
    - Uncheck **Share throughput across containers**
    - **Container id**: **OnlineUserProfile01**
    - **Partition key**: **/userId**
    - **Container throughput: Manual**
    - **RU/s: 10000**
    - **Analytical store: On**

14. Leave the remaining options with their default values

15. In the **New Container** screen, click **OK**

16. Wait until container is created. Once created go for the next task.

Task 2: Load Data into Azure Cosmos DB for NoSQL

1. In Synapse Studio, on the **Manage** hub, add a new **Linked service** for **Azure Cosmos DB for NoSQL** with the following settings:
    - **Name**: cosmosdb01
    - **Authentication type**: Account key
    - **Cosmos DB account name**: cosmosdb*xx*
    - **Database name**: CustomerProfile

2. On the **Data** hub click on + to create the **Integration dataset**:
    - **Source**: Azure Cosmos DB for NoSQL
    - **Name**: customerprofile_cosmosdb
    - **Linked service**: cosmosdb01
    - **Collection**: OnlineUserProfile01
    - **Import schema**: From connection/store

3. On the **Data** hub click on + to create the **Integration dataset**:
    - **Source**: Azure Data Lake Storage Gen2
    - **Format**: Json
    - **Name**: customerprofile_datalake
    - **Linked service**: AzureDataLakeStorage (create if not exists)
    - **File path: click on folder icon and select**: wwi-02/online-user-profiles-01/
    - **Import schema**: From connection/store

4. In Synapse Studio navigate to the **Integrate** hub.

5. In the **+** menu, select **Pipeline**.

6. Under **Activities**, expand the **Move & transform** group, then drag the **Copy data** activity onto the canvas. Set the **Name** to `Copy Cosmos DB Container` in the **Properties** blade.

7. Select the new **Copy data** activity that you added to the canvas; and on the **Source** tab beneath the canvas, select the **customerprofile_datalake** source dataset.

8. Under the source dataset

    - File path type: Wildcard file path

    - Wildcard paths **wwi-02**/ Wildcard folder path: **online-user-profile-01** / ***.json**

    - Leave rest of setting as it is.

9. Select the **Sink** tab, select the **customerprofile_cosmosdb** destination dataset.

10. Underneath the sink dataset you just added, ensure that the **Insert** write behavior is selected.

11. Under Settings tabs set Maximum data integration unit: 32
12. Select **Publish all**, then **Publish** to save the new pipeline.
13. Above the pipeline canvas, select **Add trigger**, then **Trigger now**. Select **OK** to trigger the run.
14. Navigate to the **Monitor** hub and select **Pipeline runs** and wait until the pipeline run has successfully completed.

Task 3: Querying Azure Cosmos DB with Apache Spark for Synapse Analytics

1. Navigate to the **Data** hub.

2. Select the **Linked** tab and expand the **Azure Cosmos DB** section (if this is not visible, use the ↻ button at the top right to refresh Synapse Studio), then expand the **cosmosdb01 (CustomerProfile)** linked service. Right-click the **OnlineUserProfile01** container, select **New notebook**, and then select **Load to DataFrame**.

   Notice that the **OnlineUserProfile01** container that we created has a slightly different icon than the other container. This indicates that the analytical store is enabled.

3. In the new notebook, select the **SparkPool01** Spark pool in the **Attach to** dropdown list.

4. Select **Run all**. It will take a few minutes to start the Spark session the first time.

   In the generated code within Cell 1, notice that the **spark.read** format is set to **cosmos.olap**. This instructs Synapse Link to use the container's analytical store. If we wanted to connect to the transactional store instead, like to read from the change feed or write to the container, we'd use **cosmos.oltp** instead.

   Note: You cannot write to the analytical store, only read from it. If you want to load data into the container, you need to connect to the transactional store.

   The first option configures the name of the Azure Cosmos DB linked service. The second `option` defines the Azure Cosmos DB container from which we want to read.

5. Select the **+ Code** button underneath the cell you ran. This adds a new code cell beneath the first one.

6. The DataFrame contains extra columns that we don't need. Let's remove the unwanted columns and create a clean version of the DataFrame. To do this, enter the following in the new code cell and run it:

```
unwanted_cols = {'_attachments','_etag','_rid','_self','_ts','collectionType','id'}

# Remove unwanted columns from the columns collection
cols = list(set(df.columns) - unwanted_cols)

profiles = df.select(cols)

display(profiles.limit(10))
```

   The output now only contains the columns that we want. Notice that the **preferredProducts** and **productReviews** columns contain child elements. Expand the values on a row to view them. You may recall seeing the raw JSON format in the **UserProfiles01** container within the Azure Cosmos DB Data Explorer.

7. We should know how many records we're dealing with. To do this, enter the following in a new code cell and run it:

```
profiles.count()
```

You should see a count result of 99,999.

8. We want to use the **preferredProducts** column array and **productReviews** column array for each user and create a graph of products that are from their preferred list that match with products that they have reviewed. To do this, we need to create two new DataFrames that contain flattened values from those two columns so we can join them in a later step. Enter the following in a new code cell and run it:

```
from pyspark.sql.functions import udf, explode

preferredProductsFlat=profiles.select('userId',explode('preferredProducts').alias('productI
d'))
productReviewsFlat=profiles.select('userId',explode('productReviews').alias('productReviews
'))
display(productReviewsFlat.limit(10))
```

In this cell, we imported the special PySpark explode function, which returns a new row for each element of the array. This function helps flatten the **preferredProducts** and **productReviews** columns for better readability or for easier querying.

Observe the cell output where we display the **productReviewFlat** DataFrame contents. We see a new **productReviews** column that contains the **productId** we want to match to the preferred products list for the user, as well as the **reviewText** that we want to display or save.

9. Let's look at the **preferredProductsFlat** DataFrame contents. To do this, enter the following in a new cell and **run** it:

```
display(preferredProductsFlat.limit(20))
```

Since we used the **explode** function on the preferred products array, we have flattened the column values to **userId** and **productId** rows, ordered by user.

10. Now we need to further flatten the **productReviewFlat** DataFrame contents to extract the **productReviews.productId** and **productReviews.reviewText** fields and create new rows for each data combination. To do this, enter the following in a new code cell and run it:

```
productReviews =
(productReviewsFlat.select('userId','productReviews.productId','productReviews.reviewText')
    .orderBy('userId'))

display(productReviews.limit(10))
```

In the output, notice that we now have multiple rows for each `userId`.

11. The final step is to join the **preferredProductsFlat** and **productReviews** DataFrames on the **userId** and **productId** values to build our graph of preferred product reviews. To do this, enter the following in a new code cell and run it:

```
preferredProductReviews = (preferredProductsFlat.join(productReviews,
    (preferredProductsFlat.userId == productReviews.userId) &
    (preferredProductsFlat.productId == productReviews.productId))
)

display(preferredProductReviews.limit(100))
```

    Note: You can click on the column headers in the Table view to sort the result set.

12. At the top right of the notebook, use the **Stop Session** button to stop the notebook session. Then close the notebook, discarding the changes.

Task 4: Querying Azure Cosmos DB with serverless SQL pool for Azure Synapse Analytics

1. Navigate to the **Develop** hub.
2. In the **+** menu, select **SQL script**.
3. When the script opens, in the **Properties** pane to the right, change the **Name** to `User Profile HTAP`. Then use the **Properties** button to close the pane.
4. Verify that the serverless SQL pool (**Built-in**) is selected.
5. Paste the following SQL query. In the OPENROWSET statement, replace **YOUR_ACCOUNT_NAME** with the Azure Cosmos DB account name and **YOUR_ACCOUNT_KEY** with the Azure Cosmos DB Primary Key from the **Keys** page in the Azure portal (which should still be open in another tab).

```
USE master
GO

IF DB_ID (N'Profiles') IS NULL
BEGIN
    CREATE DATABASE Profiles;
END
GO

USE Profiles
GO

DROP VIEW IF EXISTS OnlineUserProfile01;
GO

CREATE VIEW OnlineUserProfile01
AS
SELECT
    *
FROM OPENROWSET(
    'CosmosDB',
    N'account=YOUR_ACCOUNT_NAME;database=CustomerProfile;key=YOUR_ACCOUNT_KEY',
    OnlineUserProfile01
)
WITH (
    userId bigint,
    cartId varchar(50),
    preferredProducts varchar(max),
    productReviews varchar(max)
) AS profiles
CROSS APPLY OPENJSON (productReviews)
WITH (
    productId bigint,
    reviewText varchar(1000)
) AS reviews
GO
```

6. Use the **Run** button to run the query, which:

    - Creates a new serverless SQL pool database named **Profiles** if it does not exist
    - Changes the database context to the **Profiles** database.
    - Drops the **OnlineUserProfile01** view if it exists.
    - Creates a SQL view named **OnlineUserProfile01**.
    - Uses the OPENROWSET statement to set the data source type to **CosmosDB**, sets the account details, and specifies that we want to create the view over the Azure Cosmos DB analytical store container named **OnlineUserProfile01**.
    - Matches the property names in the JSON documents and applies the appropriate SQL data types. Notice that we set the **preferredProducts** and **productReviews** fields to **varchar(max)**. This is because both of these properties contain JSON-formatted data within.
    - Since the **productReviews** property in the JSON documents contain nested subarrays, the script needs to "join" the properties from the document with all elements of the array. Synapse SQL enables us to flatten the nested structure by applying the OPENJSON function on the nested array. We flatten the values within **productReviews** like we did using the Python **explode** function earlier in the Synapse Notebook.

7. Navigate to the **Data** hub.

8. Select the **Workspace** tab and expand the **Databases** group. Expand the **Profiles** SQL on-demand database (if you do not see this on the list, refresh the **Databases** list). Expand **Views**, then right-click the **OnlineUserProfile01** view, select **New SQL script**, and then **Select TOP 100 rows**.

9. Ensure the script is connected to the **Built-in** SQL pool, then run the query and view the results.

    The **preferredProducts** and **productReviews** fields are included in the view, which both contain JSON-formatted values. Notice how the CROSS APPLY OPENJSON statement in the view successfully flattened the nested subarray values in the **productReviews** field by extracting the **productId** and **reviewText** values into new fields.