P-4: Implementation

TOPIC: BRAZIL E-COMMERCE DATA ANALYSIS

TEAM: 3

Abbas Furniturewala - NUID 002193272
Shubham Idekar - NUID 002776415
Shrutika Salian - NUID 002142365
Harshit Parikh - NUID 001044838

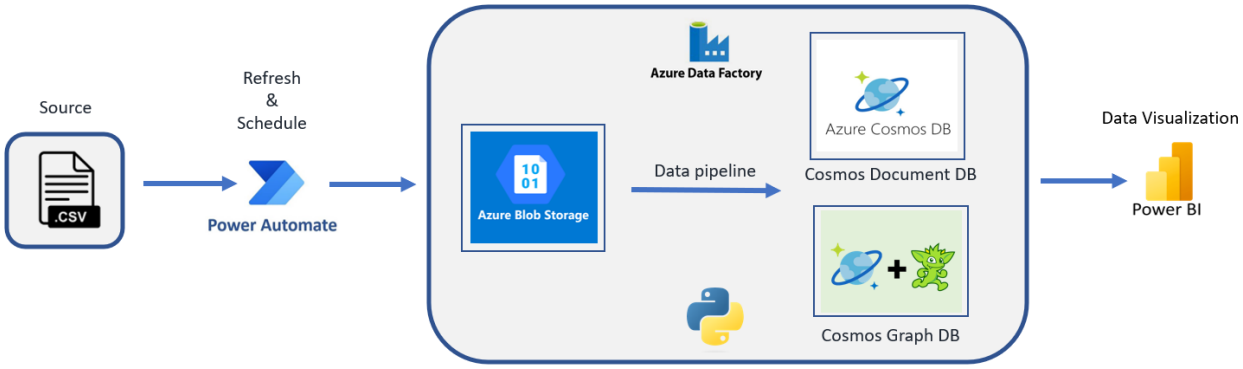
A brief description of the implementation process:

1. Connected Power Automate to Azure Blob Storage using the connection strings. As soon as a file is created in OneDrive folder, the file is fetched by Power Automate and placed into Azure Blob Storage.

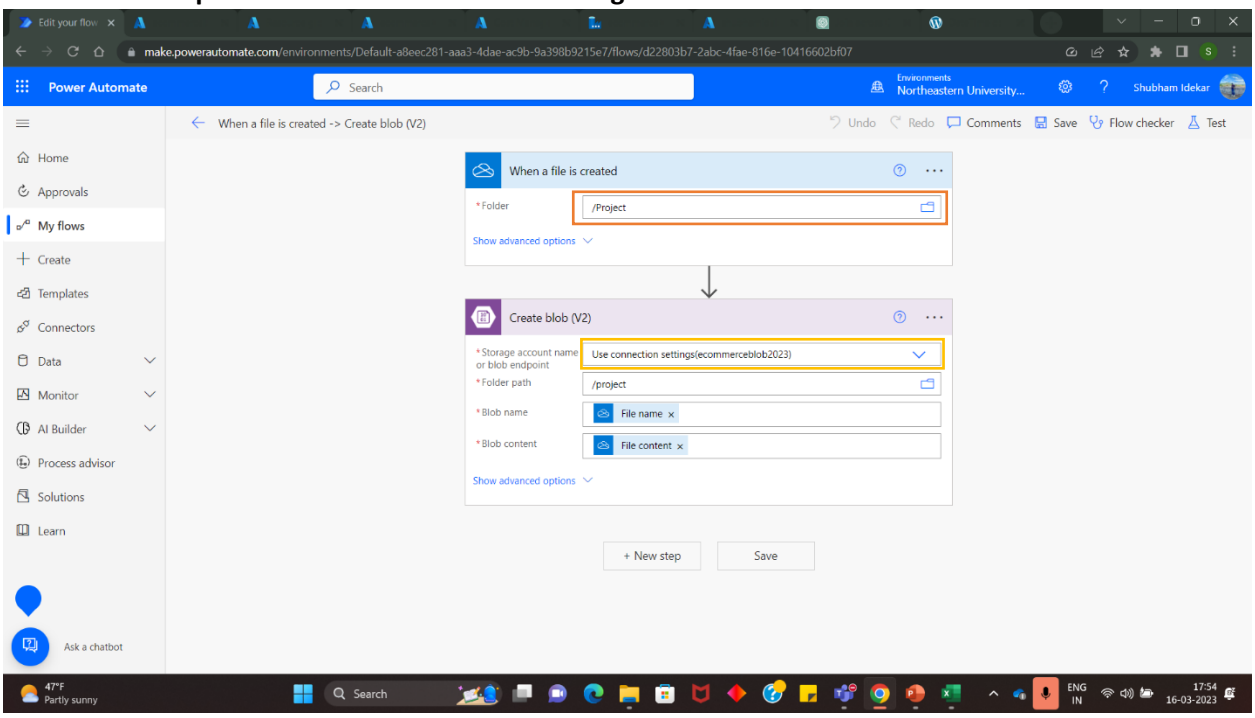
2. For Azure Cosmos Document DB implementation, the files from blob are imported into AZURE DATA FACTORY in the Dataset section. The files are cleansed, integrated, transformed, aggregated as per required hierarchy and nesting and finally, sink it into the destination Azure Cosmos Document DB.

3. For Azure Graph DB implementation, we first use Azure data factory to combine the multiple .csv files required for graph implementation into one file. We established a connection from python to our azure blob storage where the combined file is placed to directly fetch from there. Then, established a connection from python to Azure Cosmos Graph DB using Gremlin API. Once the connection was established, we created the vertices and edges using the Gremlin API, ingesting data into Azure Cosmos Graph DB.

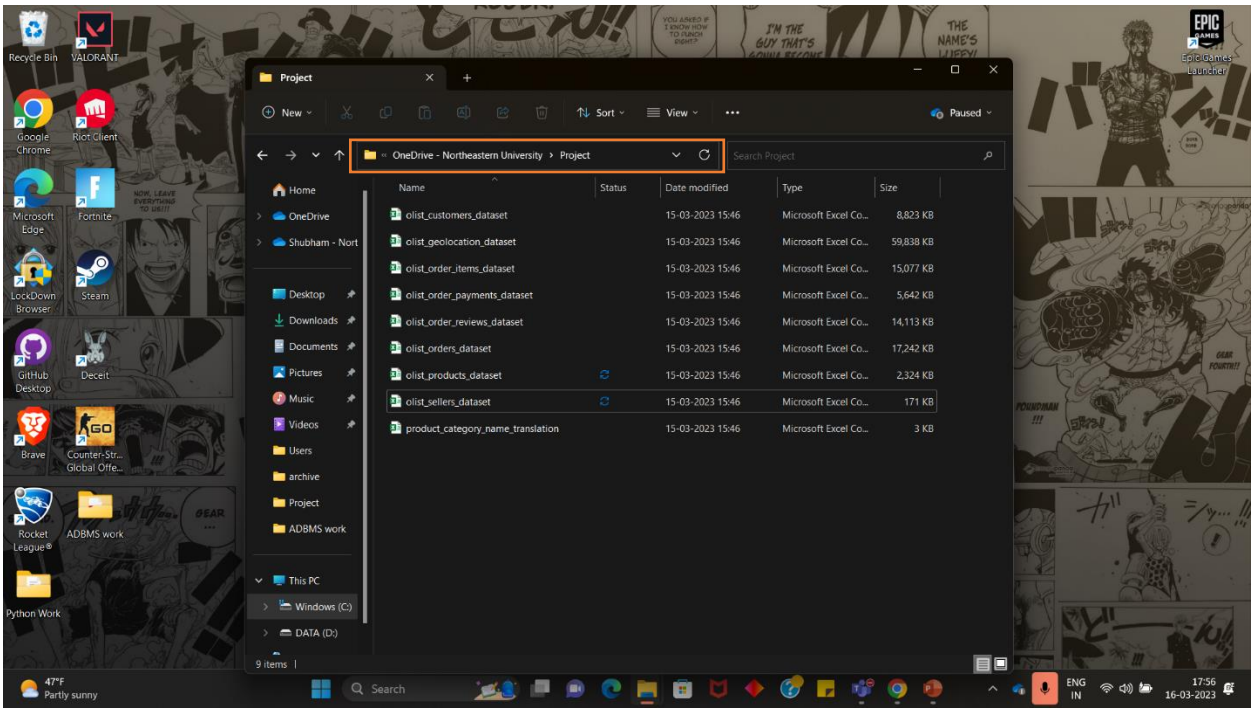
Revised Data Architecture Diagram:



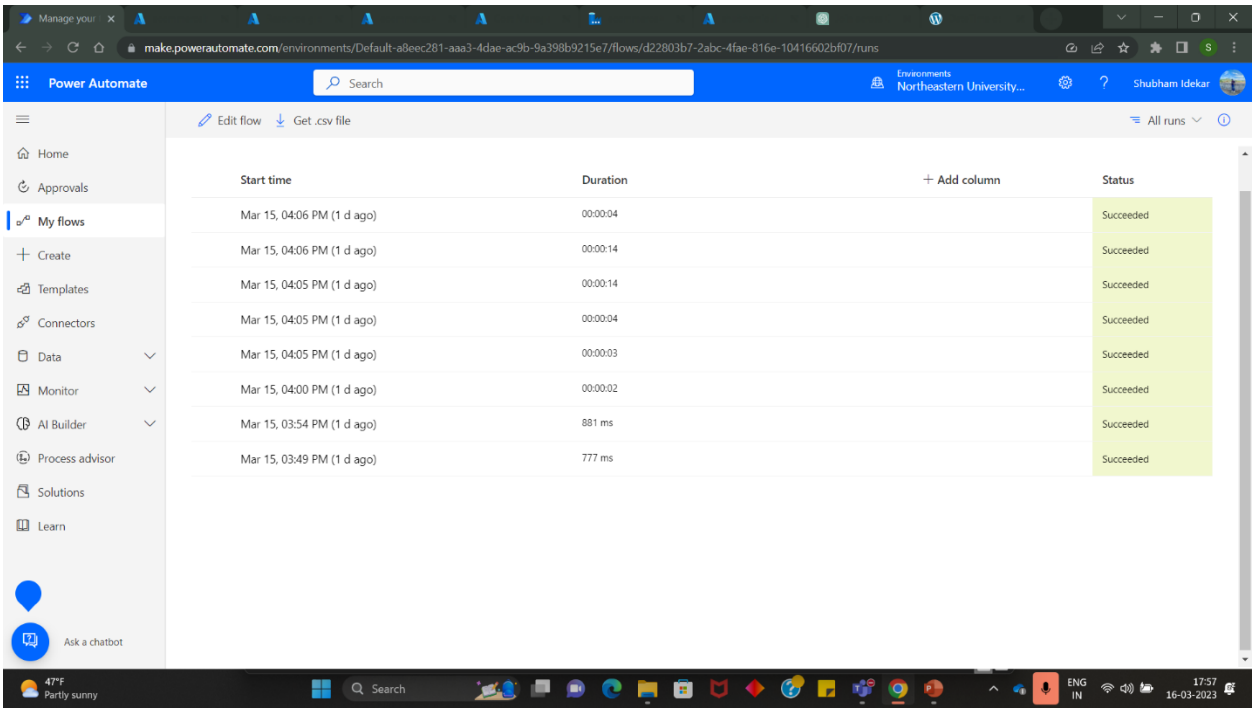
Using **Power Automate Automated Cloud Flow** to **fetch files from OneDrive folder** whenever a **new file is created** and **upload the files to Azure Blob Storage**:



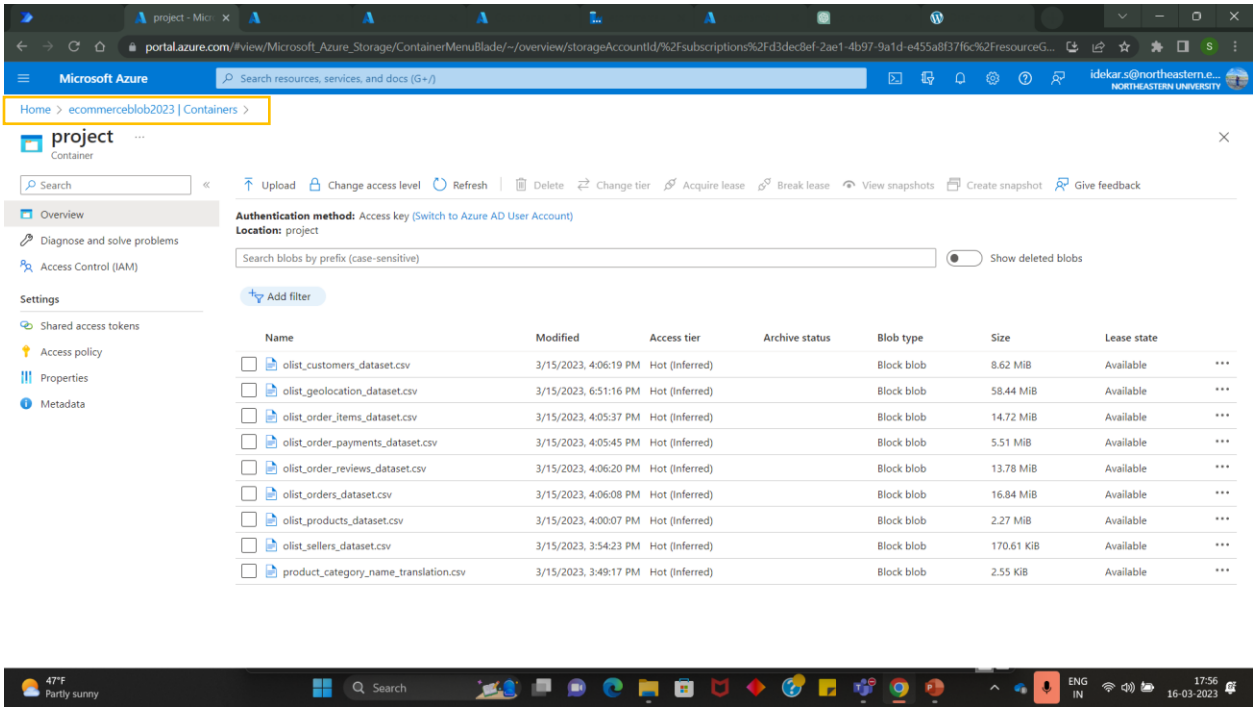
Placing .csv files (dataset) into the OneDrive folder:



Successful execution of Power Automate flow:

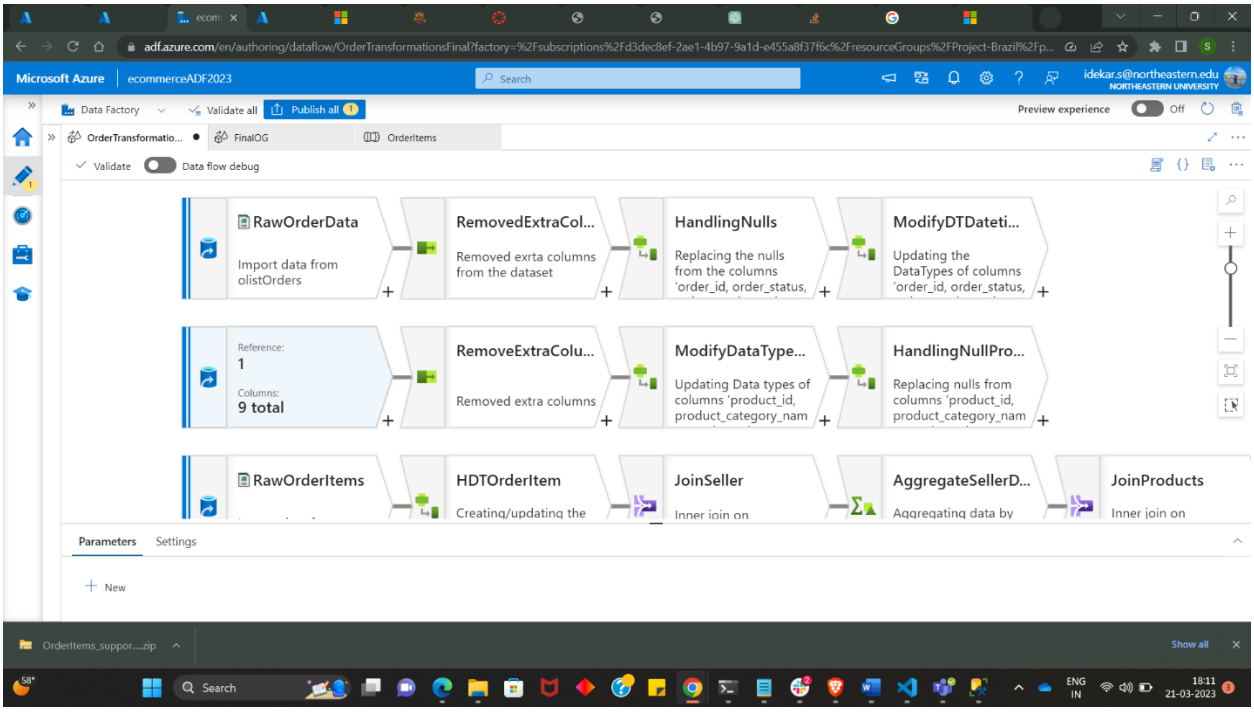


All files placed into Azure Blob Storage automatically once created in OneDrive Folder:



AZURE COSMOS DOCUMENT DB implementation:

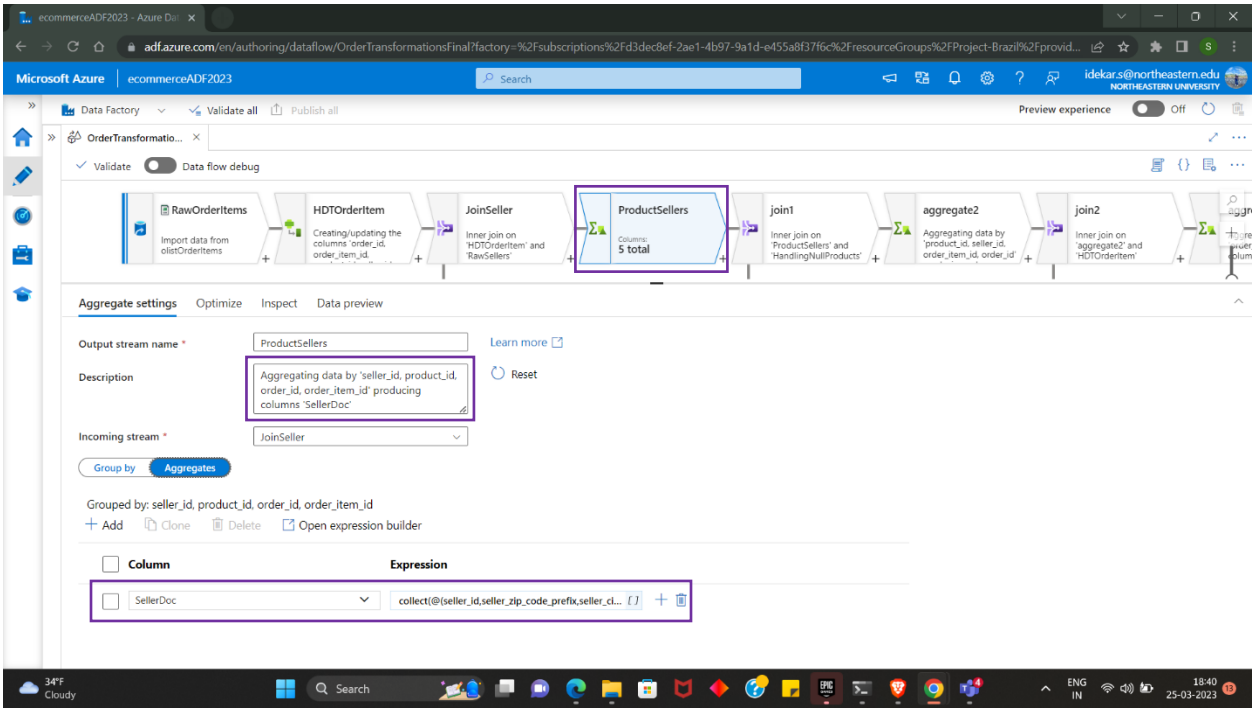
Created a **Dataflow** in **Azure Data Factory** to perform some **data cleansing and transformation processes** on .csv files like: **Removing Extra Columns, Handling Null Values in columns, Modifying the datetime, timestamp and other data types to required format and data type:**



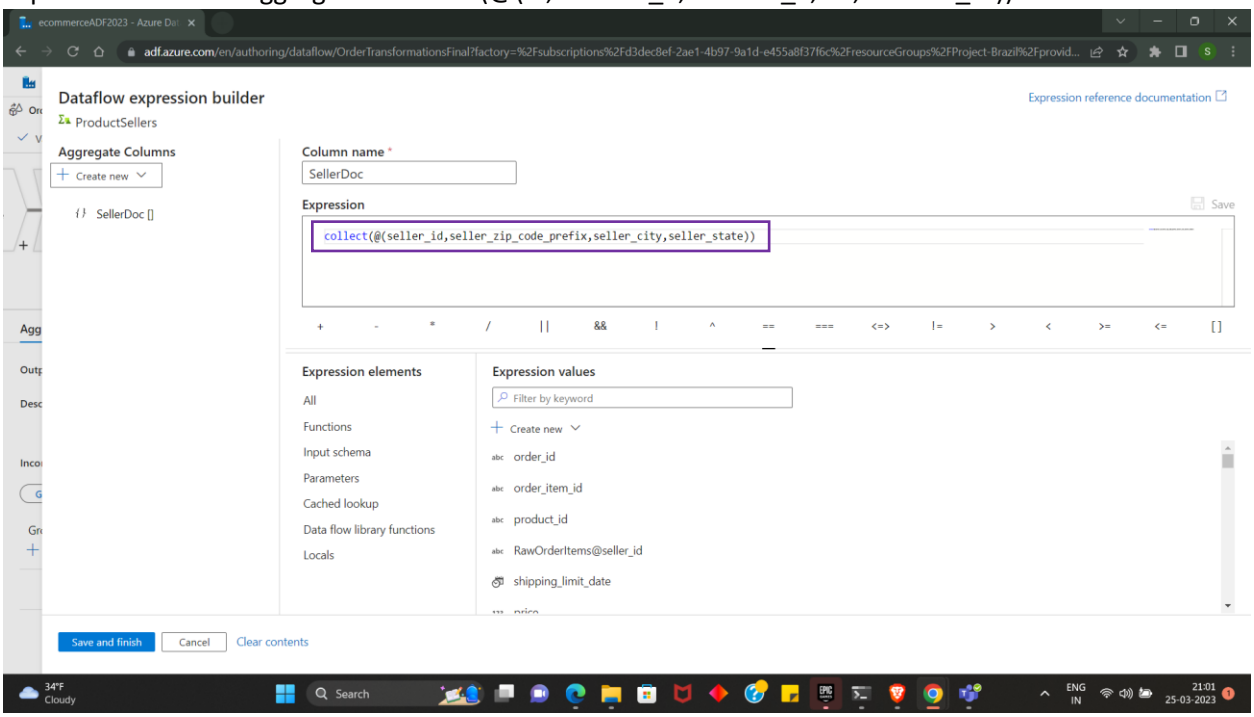
Applying **joins to combine data from multiple .csvs**.

After join at each step **Aggregating by id**, to achieve the format required for Document DB:

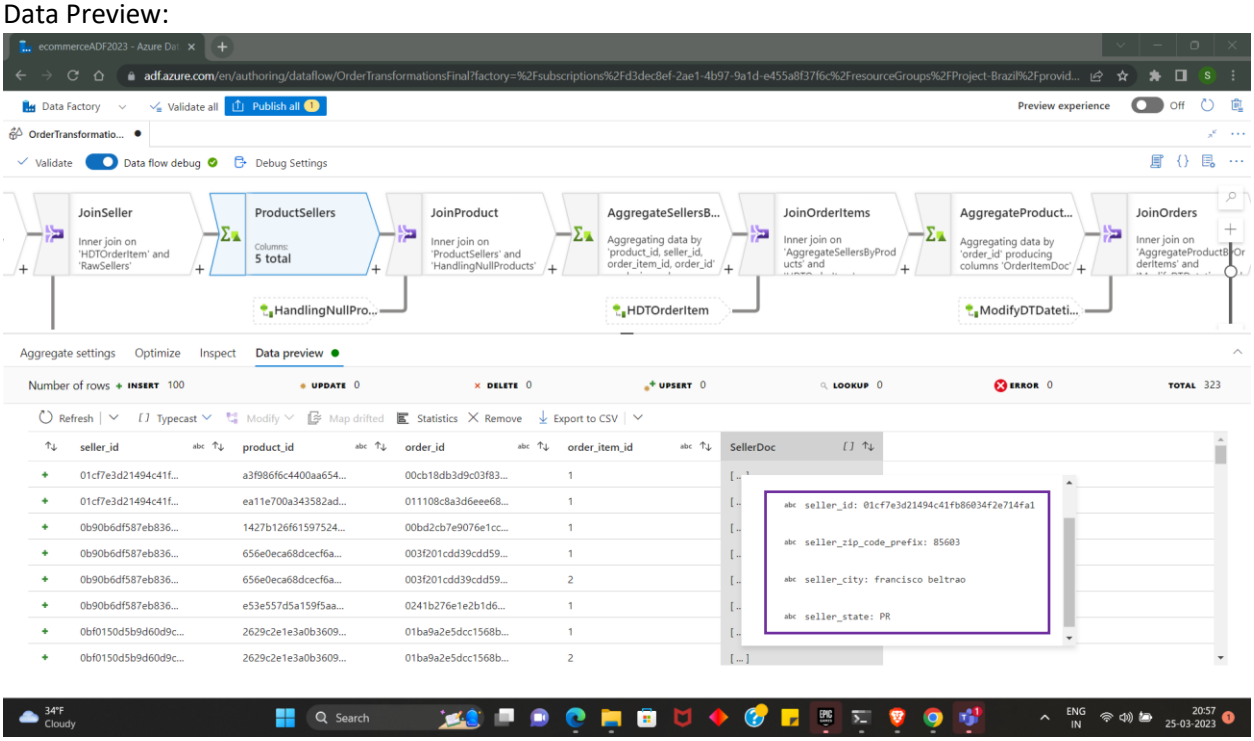
Below, aggregated by seller_id:



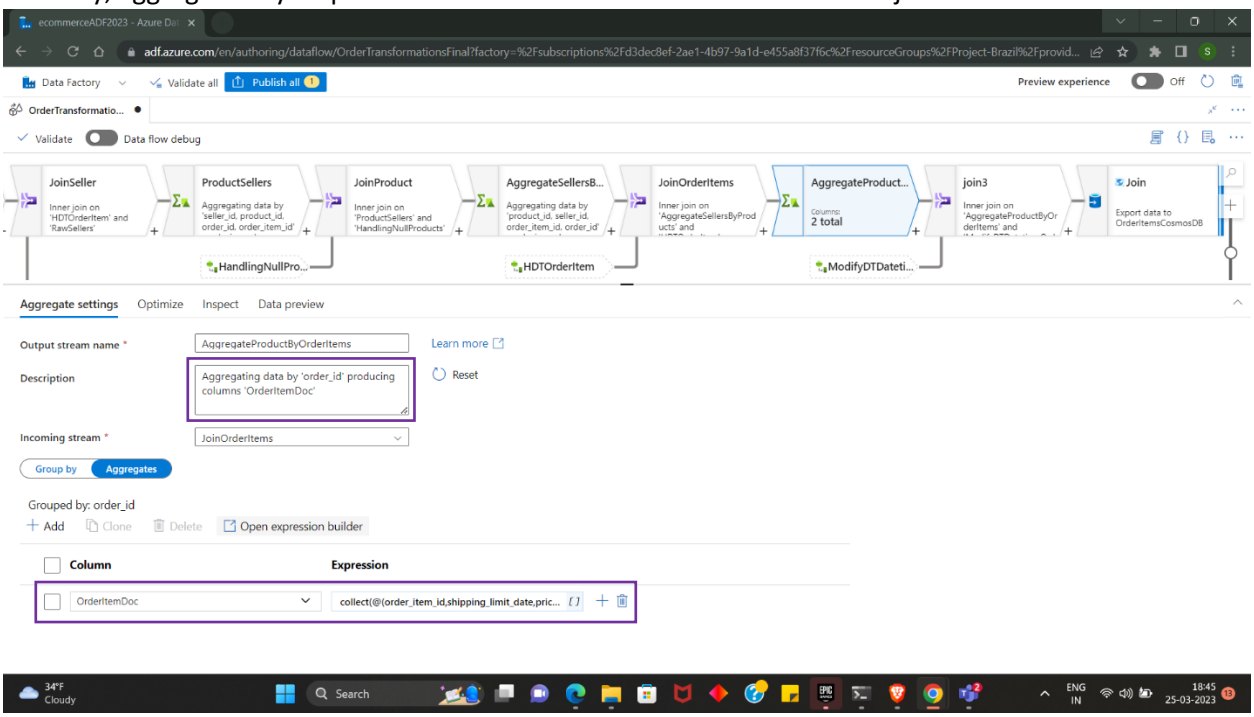
Expression used for Aggregation: collect(@(id, column_1, column_2,….., column_N))



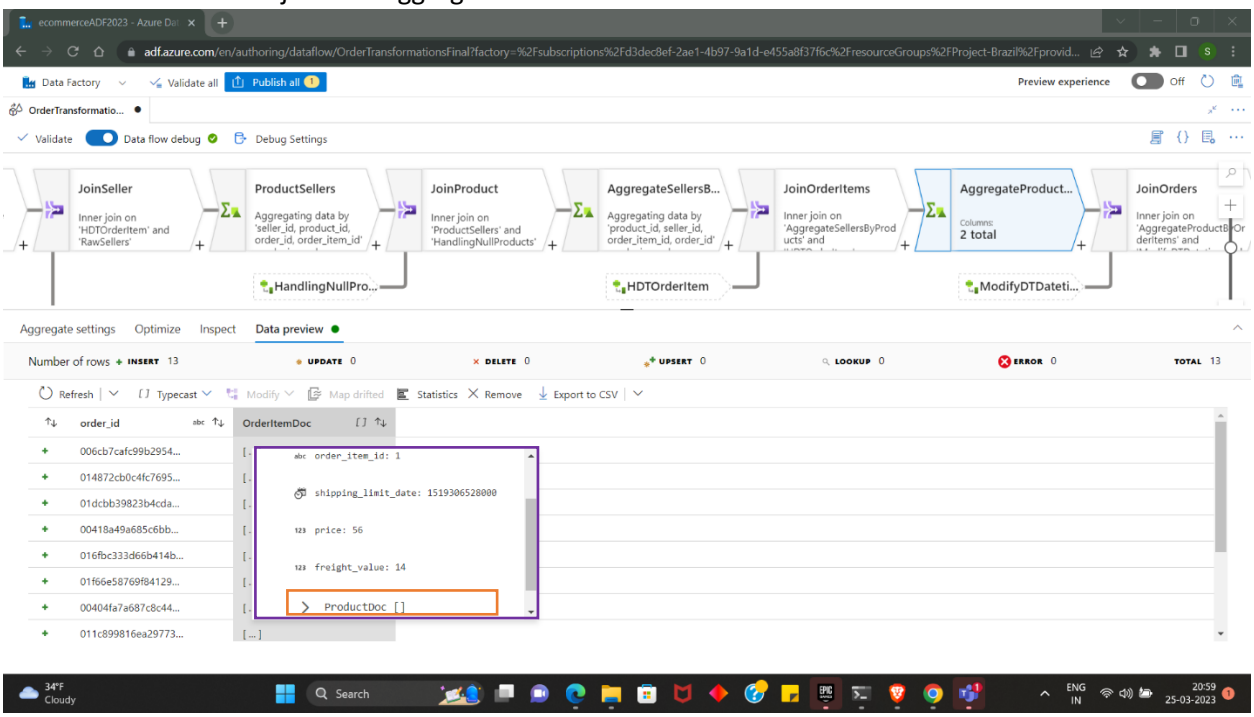
Data Preview:



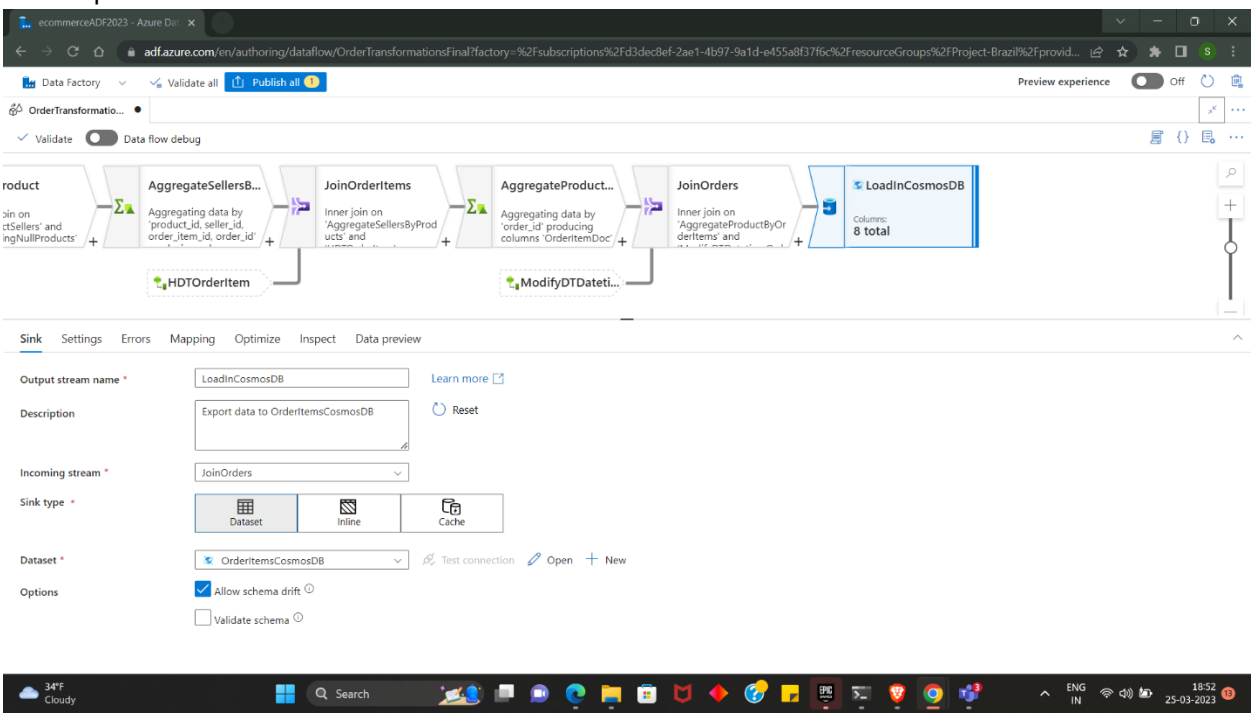
Similarly, aggregated by respective ids to achieve the nested document db json format:



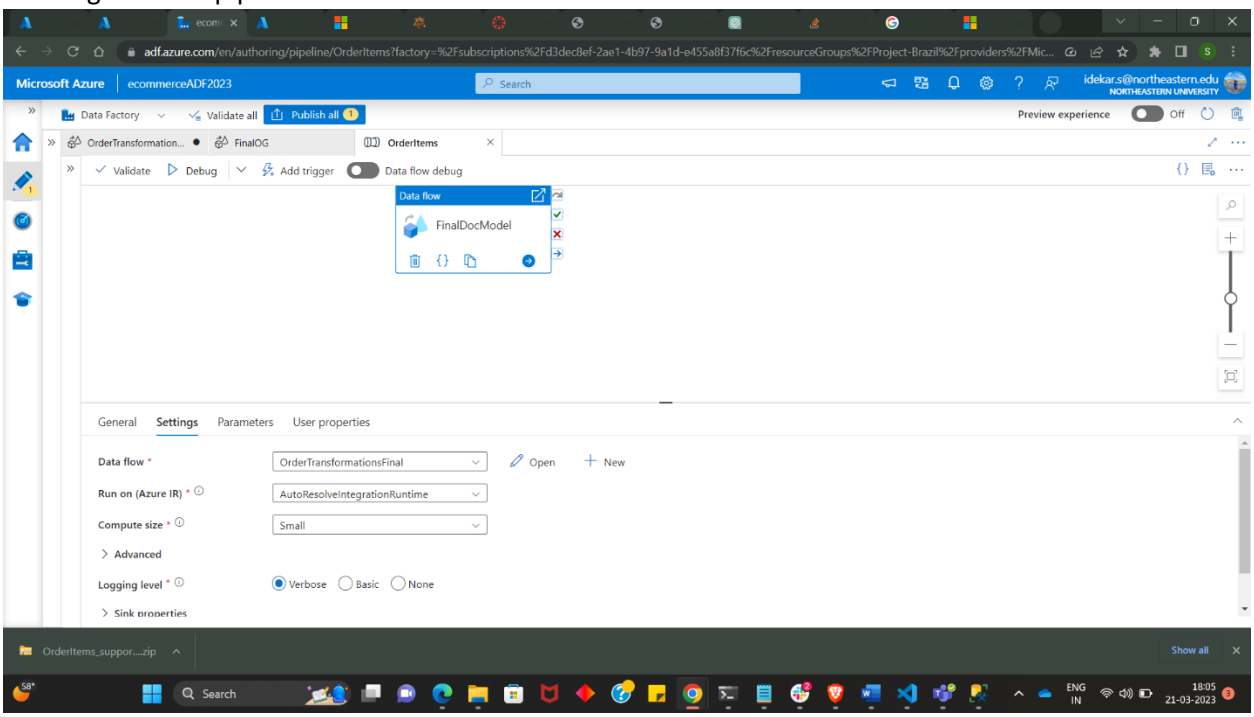
Data Preview after last join and aggregation:



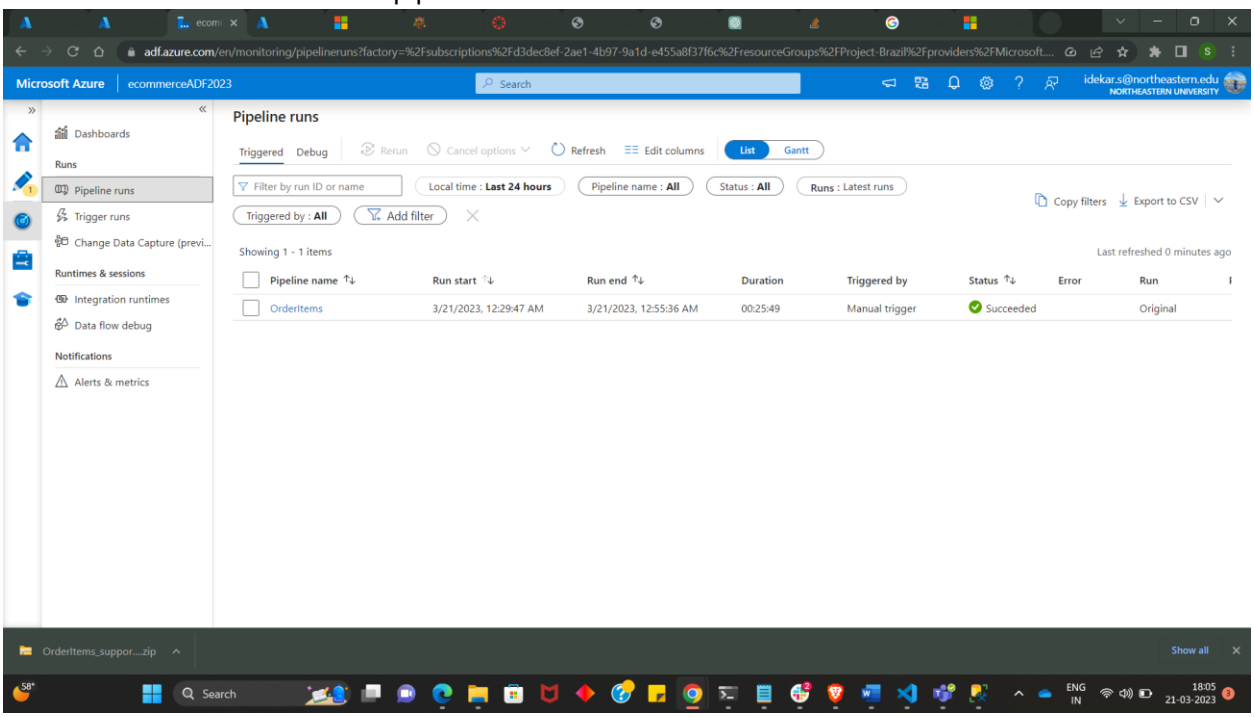
Last step to sink the data into Azure Cosmos Document DB:



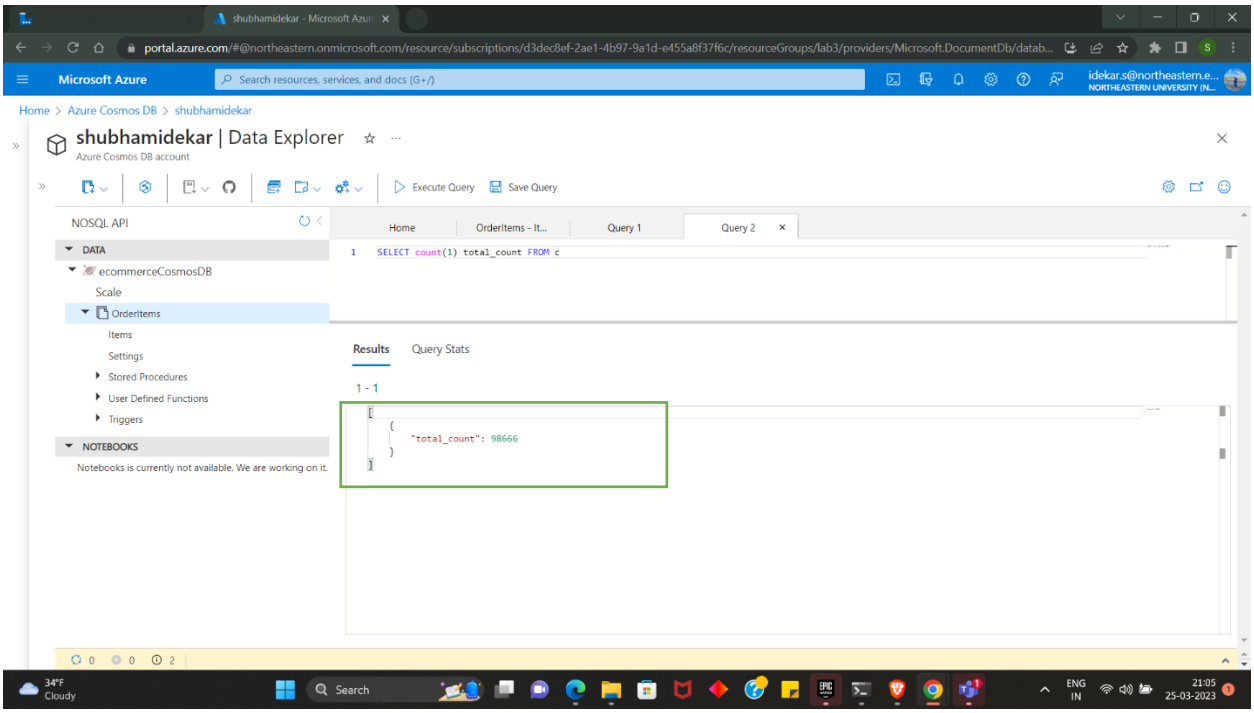
Running the data pipeline:



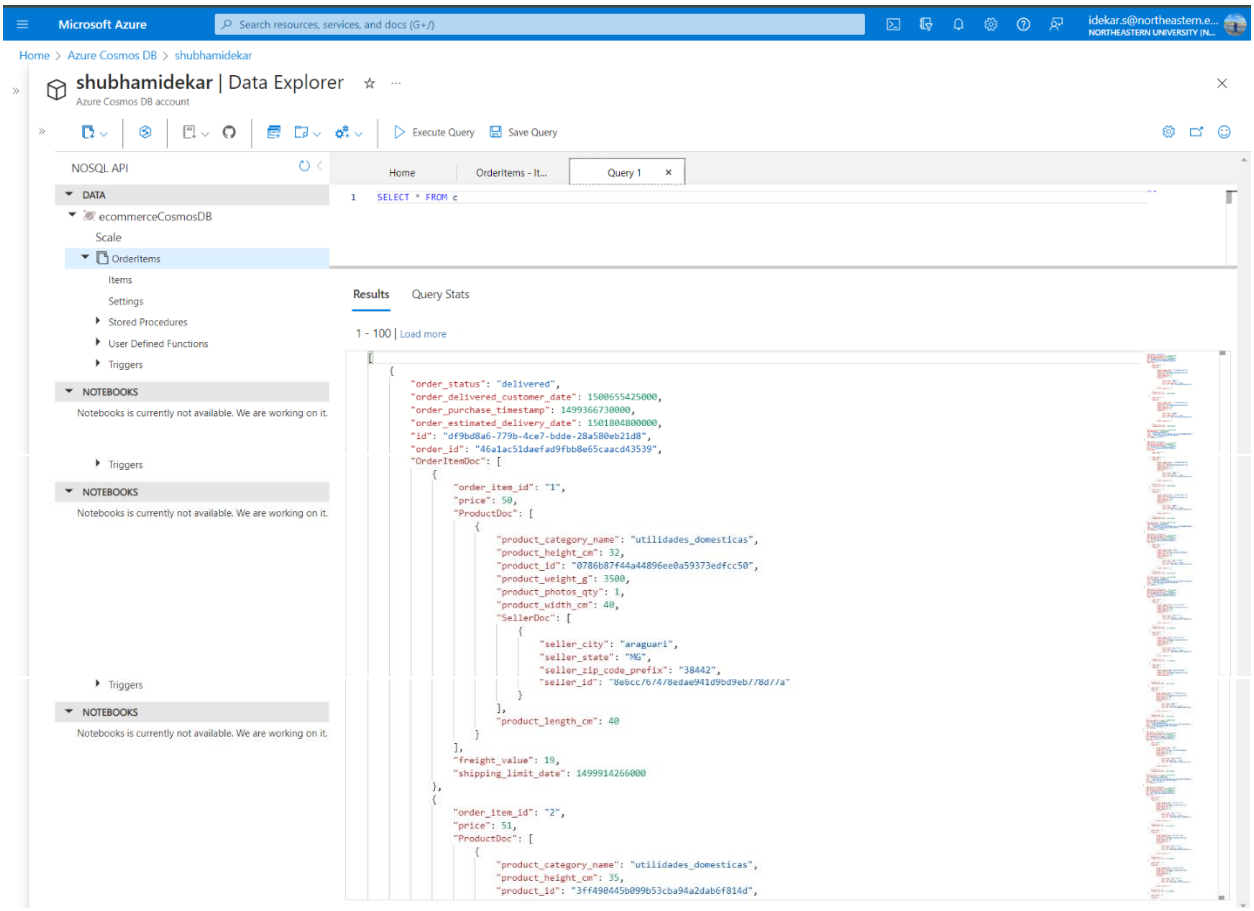
Successful execution of the data pipeline:



Count of Data loaded in Azure Cosmos Document DB:



Document DB loaded in Azure:

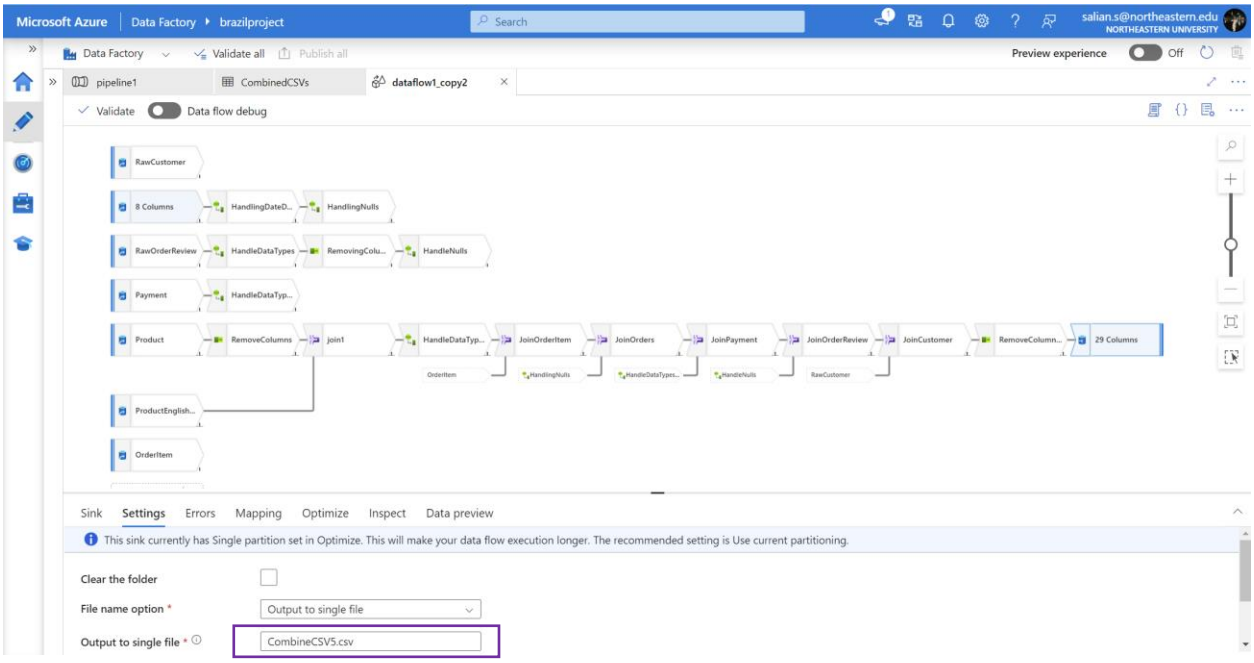


Revised Document DB Data model: (Combined Order, OrderItems, Products, Seller and Geolocation into one Document DB as per querying needs)

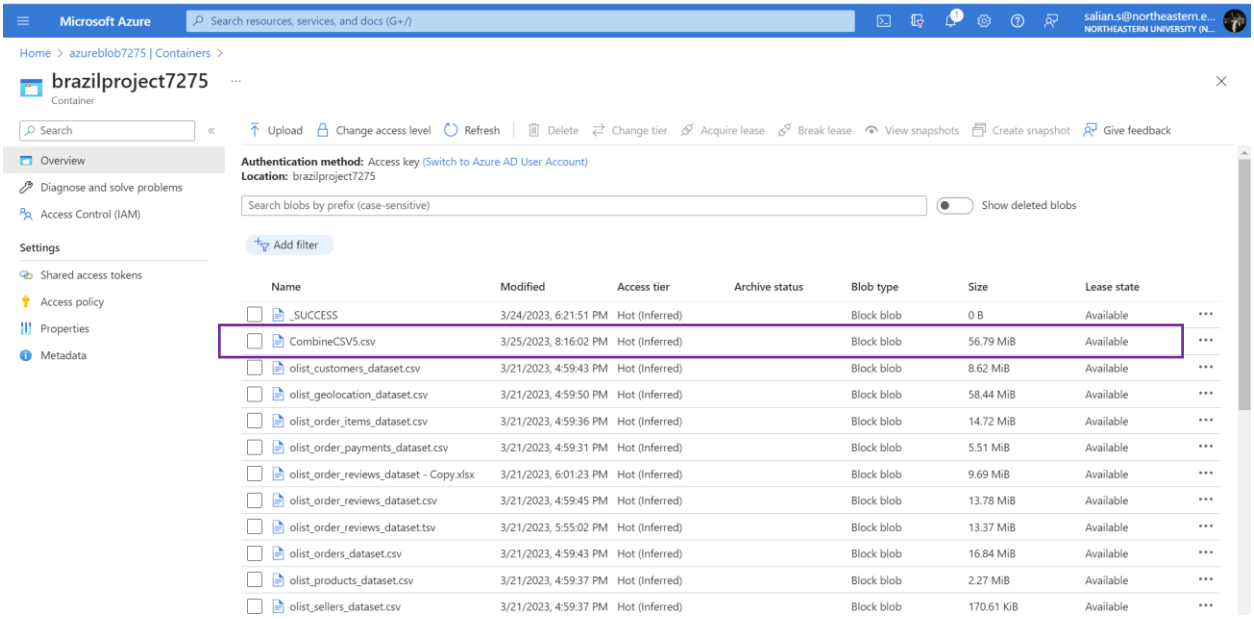
```
{
    "order_status",
    "order_delivered_customer_date",
    "order_purchase_timestamp",
    "order_estimated_delivery_date",
    "id",
    "order_id",
      "OrderItemDoc": [
        { "order_item_id",
          "price",
          "ProductDoc": [
              { "product_category_name",
                  "product_height_cm",
                  "product_id",
                  "product_weight_g",
                  "product_photos_qty",
                  "product_width_cm",
                  "product_length_cm",
                  "freight_value",
                  "shipping_limit_date",
                  "SellerDoc": [
                      {"seller_city",
                       "seller_state",
                       "seller_zip_code_prefix",
                       "seller_id"
                            }
                       ]
                       }
                       ]
                     }
```

AZURE COSMOS GRAPH DOCUMENT DB implementation using GREMLIN API:

Created a **Dataflow** in **Azure Data Factory** to perform some **data cleansing and transformation processes** on .csv files like: **Removing Extra Columns, Handling Null Values in columns, Modifying the datetime, timestamp and other data types to required format and data type, combining all .csvs files into one file:**



Combined .csv file stored in Azure blob storage:



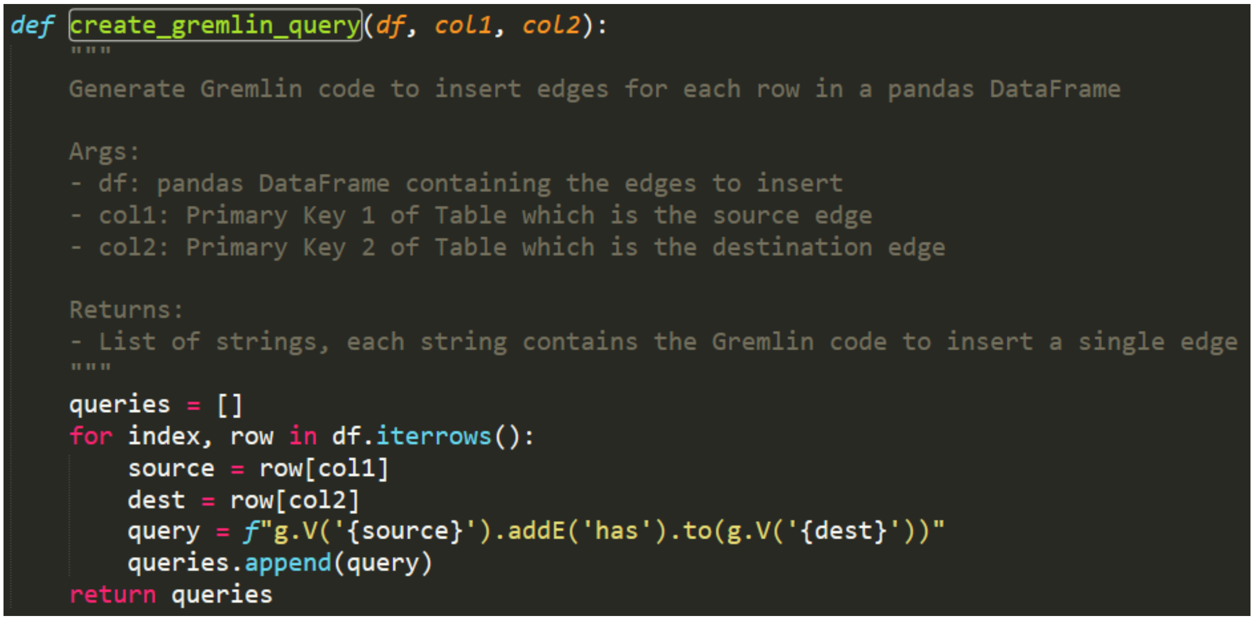
Function for automatically identifying primary key, properties and creating Vertices with it's properties from dataset:

```python
def generate_gremlin_code(df, vertex_label: str, pk_column: str) -> List[str]:
    """
    Generate Gremlin code to insert vertices for each row in a pandas DataFrame

    Args:
    - df: pandas DataFrame containing the data to insert
    - vertex_label: str, the label of the vertex to create
    - pk_column: str, the name of the column containing the primary key

    Returns:
    - List of strings, each string contains the Gremlin code to insert a single vertex
    """
    gremlin_code = []
    for i, row in df.iterrows():
        pk_value = str(row[pk_column])
        properties = []
        for col, value in row.iteritems():
            if col != pk_column:
                properties.append(f".property('{col}', '{value}')")
        properties = "".join(properties)
        vertex_code = f"g.addV('{vertex_label}').property('id', '{pk_value}').property('_id', '{pk_value}'){properties}"
        gremlin_code.append(vertex_code)
    return gremlin_code
```

Function for inserting Vertices:

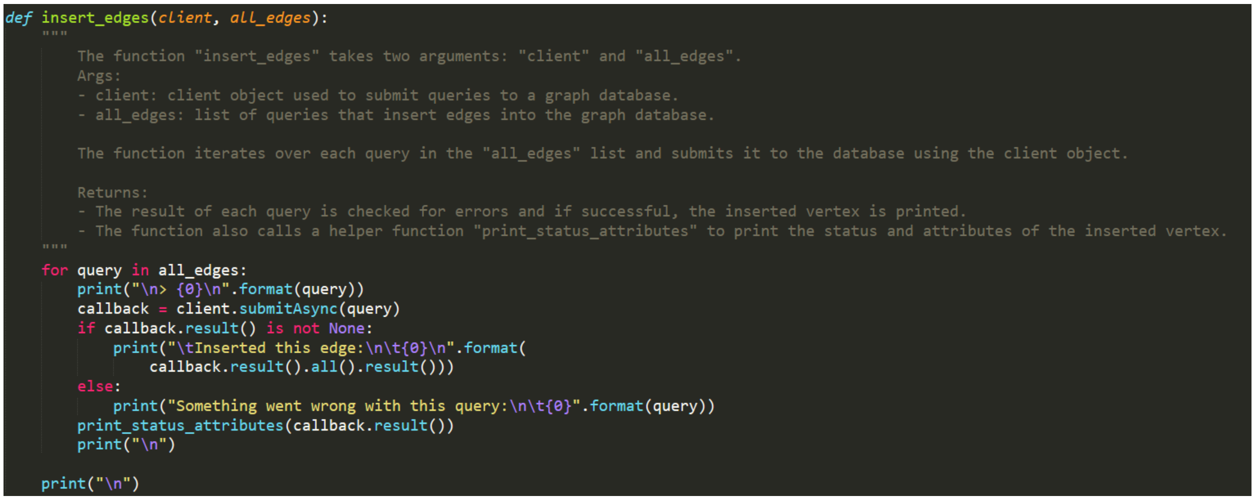
```python
def insert_vertices(client, all_vertices):
    """
        The function "insert_vertices" takes two arguments: "client" and "all_vertices".
        Args:
        - client: client object used to submit queries to a graph database.
        - all_vertices: list of queries that insert vertices into the graph database.

        The function iterates over each query in the "all_vertices" list and submits it to the database using the client object.

        Returns:
        - The result of each query is checked for errors and if successful, the inserted vertex is printed.
        - The function also calls a helper function "print_status_attributes" to print the status and attributes of the inserted vertex.
    """
    for query in all_vertices:
        print("\n> {0}\n".format(query))
        callback = client.submitAsync(query)
        if callback.result() is not None:
            print("\tInserted this vertex:\n\t{0}".format(
                callback.result().all().result()))
        else:
            print("Something went wrong with this query: {0}".format(query))
        print("\n")
        print_status_attributes(callback.result())
        print("\n")

    print("\n")
```

Function for automatically identifying source & destination and creating Edges from dataset:



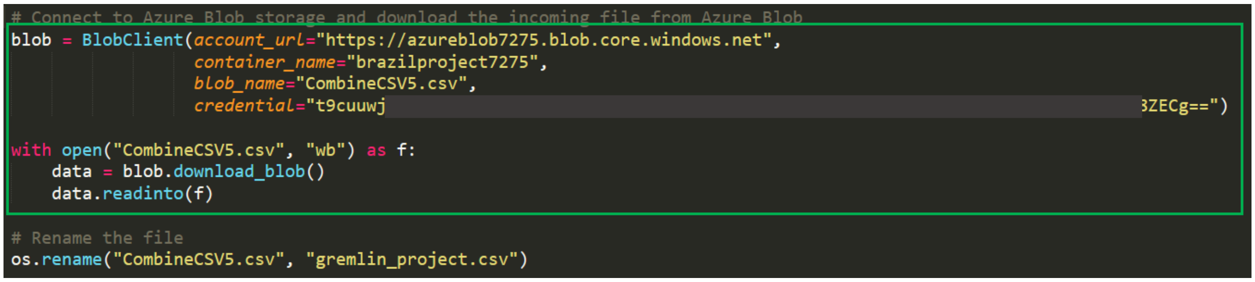
```python
def create_gremlin_query(df, col1, col2):
    """
    Generate Gremlin code to insert edges for each row in a pandas DataFrame

    Args:
    - df: pandas DataFrame containing the edges to insert
    - col1: Primary Key 1 of Table which is the source edge
    - col2: Primary Key 2 of Table which is the destination edge

    Returns:
    - List of strings, each string contains the Gremlin code to insert a single edge
    """
    queries = []
    for index, row in df.iterrows():
        source = row[col1]
        dest = row[col2]
        query = f"g.V('{source}').addE('has').to(g.V('{dest}'))"
        queries.append(query)
    return queries
```

Function for inserting Edges:



```python
def insert_edges(client, all_edges):
    """
        The function "insert_edges" takes two arguments: "client" and "all_edges".
        Args:
        - client: client object used to submit queries to a graph database.
        - all_edges: list of queries that insert edges into the graph database.

        The function iterates over each query in the "all_edges" list and submits it to the database using the client object.

        Returns:
        - The result of each query is checked for errors and if successful, the inserted vertex is printed.
        - The function also calls a helper function "print_status_attributes" to print the status and attributes of the inserted vertex.
    """
    for query in all_edges:
        print("\n> {0}\n".format(query))
        callback = client.submitAsync(query)
        if callback.result() is not None:
            print("\tInserted this edge:\n\t{0}\n".format(
                callback.result().all().result()))
        else:
            print("Something went wrong with this query:\n\t{0}".format(query))
        print_status_attributes(callback.result())
        print("\n")

    print("\n")
```

Establishing connection to Azure Blob Storage from python to fetch the combined csv dataset file:



```python
# Connect to Azure Blob storage and download the incoming file from Azure Blob
blob = BlobClient(account_url="https://azureblob7275.blob.core.windows.net",
                  container_name="brazilproject7275",
                  blob_name="CombineCSV5.csv",
                  credential="t9cuuwj[redacted]ZECg==")

with open("CombineCSV5.csv", "wb") as f:
    data = blob.download_blob()
    data.readinto(f)

# Rename the file
os.rename("CombineCSV5.csv", "gremlin_project.csv")
```

Establishing connection to Azure Cosmos DB from python using Gremlin API:

```
try:
    client = client.Client('wss://azure-gremlin-hp.gremlin.cosmos.azure.com:443/', 'g',
                           username="/dbs/gremlin_test/colls/Person1",
                           password="vor028                                                                    ==",
                           message_serializer=serializer.GraphSONSerializersV2d0()
                           )

    print("Welcome to Azure Cosmos DB + Gremlin on Python!")

    print("Let's drop everything first!")
    cleanup_graph(client)
```

Calling the functions to insert Vertices and Edges:

```
insert_vertices(client, generate_gremlin_code(order, 'order', 'order_id'))
insert_vertices(client, generate_gremlin_code(product, 'product', 'product_id'))
insert_vertices(client, generate_gremlin_code(order_review, 'order_review', 'order_review_SK'))
insert_vertices(client, generate_gremlin_code(customer, 'customer', 'customer_id'))
insert_vertices(client, generate_gremlin_code(payment, 'payment', 'payment_SK'))

insert_edges(client, create_gremlin_query(order_product, 'order_id', 'product_id')) # Order and Product
insert_edges(client, create_gremlin_query(order_payment, 'order_id', 'payment_SK')) # Order and Payment
insert_edges(client, create_gremlin_query(order_customer, 'order_id', 'customer_id')) # Order and Customer
insert_edges(client, create_gremlin_query(order_order_review, 'order_id', 'order_review_SK')) # Order and Order Review
```

Executing the script in command prompt:

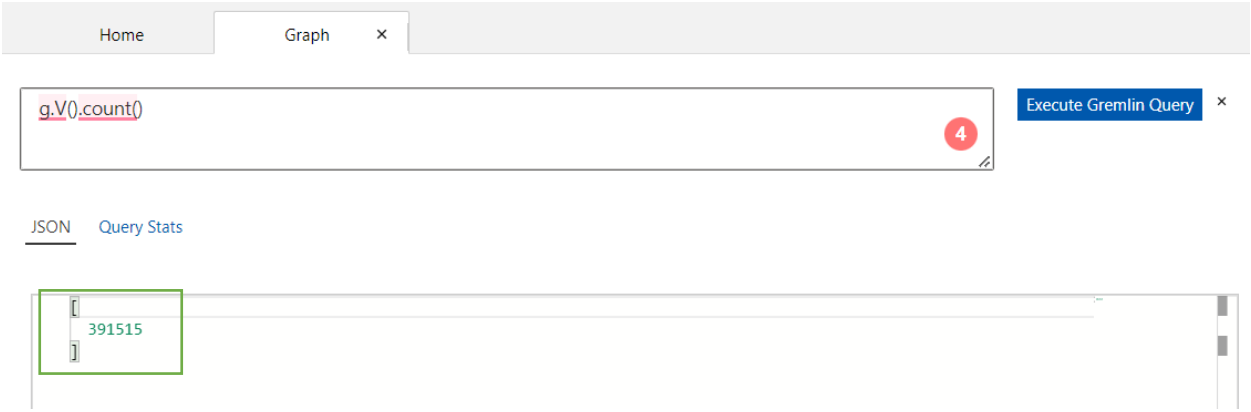
Vertices being successfully created:

```
C:\Windows\System32\cmd.exe

        Response status_attributes:
        {'x-ms-status-code': 200, 'x-ms-activity-id': '724f6823-5caa-43f5-ac6c-ff5b2507ae66', 'x-ms-request-charge': 16.95, 'x-ms-total-request-charge': 16.95, 'x-ms-server-time-ms': 7.1836,
'x-ms-total-server-time-ms': 7.1836}



> g.addV('order').property('id', '04bef5ebaab368e72d66decdad88fddf').property('_id', '04bef5ebaab368e72d66decdad88fddf').property('customer_id', '55ca07ff32dc30f00ded4c99f8ea9124').property('
order_status', 'delivered').property('order_purchase_timestamp', '7/6/2017 16:54').property('order_approved_at', '7/6/2017 17:10').property('order_delivered_carrier_date', '7/19/2017 12:57').
property('order_delivered_customer_date', '7/19/2017 12:57').property('order_estimated_delivery_date', '8/1/2017 0:00')

        Inserted this vertex:
        [{'id': '04bef5ebaab368e72d66decdad88fddf', 'label': 'order', 'type': 'vertex', 'properties': {'_id': [{'id': '04bef5ebaab368e72d66decdad88fddf|_id', 'value': '04bef5ebaab368e72d66dec
dad88fddf'}], 'customer_id': [{'id': '2591a624-7357-4de9-9905-8c6d05c554b6', 'value': '55ca07ff32dc30f00ded4c99f8ea9124'}], 'order_status': [{'id': 'e04b80e1-00a4-4324-9a45-866a47553511', 'va
lue': 'delivered'}], 'order_purchase_timestamp': [{'id': 'f5c31864-30f4-4b81-938b-de4aceec8d6d', 'value': '7/6/2017 16:54'}], 'order_approved_at': [{'id': '5465cbe6-4f82-472d-95a0-21af74142a1
3', 'value': '7/6/2017 17:10'}], 'order_delivered_carrier_date': [{'id': '72ac7fbb-cf2b-4169-b797-cd17603c4766', 'value': '7/19/2017 12:57'}], 'order_delivered_customer_date': [{'id': '20d95f
80-4ed0-43d3-a3b8-b6f8aab57800', 'value': '7/19/2017 12:57'}], 'order_estimated_delivery_date': [{'id': 'c097019e-c2fb-4f6b-91a7-a5a9e1813b2f', 'value': '8/1/2017 0:00'}]}]}]


        Response status_attributes:
        {'x-ms-status-code': 200, 'x-ms-activity-id': 'b1c628f0-dd61-458d-9d65-95676d08fa88', 'x-ms-request-charge': 16.95, 'x-ms-total-request-charge': 16.95, 'x-ms-server-time-ms': 6.547, '
x-ms-total-server-time-ms': 6.547}



> g.addV('order').property('id', '04e00ba23c33890eaee39b02e8185cc2').property('_id', '04e00ba23c33890eaee39b02e8185cc2').property('customer_id', '291fd0287b7cfd75702b9be21f23f1e3').property('
order_status', 'delivered').property('order_purchase_timestamp', '7/25/2018 21:36').property('order_approved_at', '7/25/2018 21:45').property('order_delivered_carrier_date', '7/30/2018 19:03'
).property('order_delivered_customer_date', '7/30/2018 19:03').property('order_estimated_delivery_date', '8/22/2018 0:00')

        Inserted this vertex:
        [{'id': '04e00ba23c33890eaee39b02e8185cc2', 'label': 'order', 'type': 'vertex', 'properties': {'_id': [{'id': '04e00ba23c33890eaee39b02e8185cc2|_id', 'value': '04e00ba23c33890eaee39b0
2e8185cc2'}], 'customer_id': [{'id': '0e58ec54-e76d-41e5-8c6b-9eb7fae31769', 'value': '291fd0287b7cfd75702b9be21f23f1e3'}], 'order_status': [{'id': '06695dd6-8f39-481c-b1c1-616bf3c18ec8', 'va
lue': 'delivered'}], 'order_purchase_timestamp': [{'id': '47fb80c4-8bbf-4c58-8c2d-e71cf94ad06b', 'value': '7/25/2018 21:36'}], 'order_approved_at': [{'id': 'f0c8baaf-d1bc-48fd-bc24-4eb18a904e
6c', 'value': '7/25/2018 21:45'}], 'order_delivered_carrier_date': [{'id': '4f1dbbb7-fe08-4ba5-bd24-10186f06c88b', 'value': '7/30/2018 19:03'}], 'order_delivered_customer_date': [{'id': 'a3e2
c7ff-be7d-4fae-85ca-d6eb51b6109f', 'value': '7/30/2018 19:03'}], 'order_estimated_delivery_date': [{'id': '4ce4a4d3-17c5-4356-a60d-dcf57e672f65', 'value': '8/22/2018 0:00'}]}]}]


        Response status_attributes:
        {'x-ms-status-code': 200, 'x-ms-activity-id': '026cc0c4-101c-40ed-860e-a464e5686e83', 'x-ms-request-charge': 16.95, 'x-ms-total-request-charge': 16.95, 'x-ms-server-time-ms': 6.5956,
'x-ms-total-server-time-ms': 6.5956}
```

Edges being sucessfully created:

```
Command Prompt
> g.V('cd725c256dd60e25a721e1fe8cccdaa6').addE('has').to(g.V('dc9674016642e9b975996058421c2976_4.0'))

        Inserted this edge:
        [{'id': '14590dc2-041f-4977-acd2-72df2d3e8335', 'label': 'has', 'type': 'edge', 'inVLabel': 'order_review', 'out
VLabel': 'order', 'inV': 'dc9674016642e9b975996058421c2976_4.0', 'outV': 'cd725c256dd60e25a721e1fe8cccdaa6'}]

        Response status_attributes:
        {'x-ms-status-code': 200, 'x-ms-activity-id': '05642959-9c74-4eb4-853c-49c7753c5abe', 'x-ms-request-charge': 14.
420000000000002, 'x-ms-total-request-charge': 14.420000000000002, 'x-ms-server-time-ms': 13.2569, 'x-ms-total-server-tim
e-ms': 13.2569}



> g.V('ce08f40b106df255b226e0cf2df6a11f').addE('has').to(g.V('464df9bea60d90a4b8eae55e60759b20_5.0'))

        Inserted this edge:
        [{'id': '77c46049-e339-49d4-a332-6d880f036df1', 'label': 'has', 'type': 'edge', 'inVLabel': 'order_review', 'out
VLabel': 'order', 'inV': '464df9bea60d90a4b8eae55e60759b20_5.0', 'outV': 'ce08f40b106df255b226e0cf2df6a11f'}]

        Response status_attributes:
        {'x-ms-status-code': 200, 'x-ms-activity-id': '4e42a54e-c920-4964-abc1-03893f21cf73', 'x-ms-request-charge': 14.
420000000000002, 'x-ms-total-request-charge': 14.420000000000002, 'x-ms-server-time-ms': 13.359, 'x-ms-total-server-time
-ms': 13.359}



> g.V('ce11340d59020c4c840f12096c81b609').addE('has').to(g.V('006862652d256edde8f62c5d7792ee50_5.0'))

        Inserted this edge:
        [{'id': 'a4ed3729-3d22-44df-b34e-652f8a1b2042', 'label': 'has', 'type': 'edge', 'inVLabel': 'order_review', 'out
VLabel': 'order', 'inV': '006862652d256edde8f62c5d7792ee50_5.0', 'outV': 'ce11340d59020c4c840f12096c81b609'}]

        Response status_attributes:
```

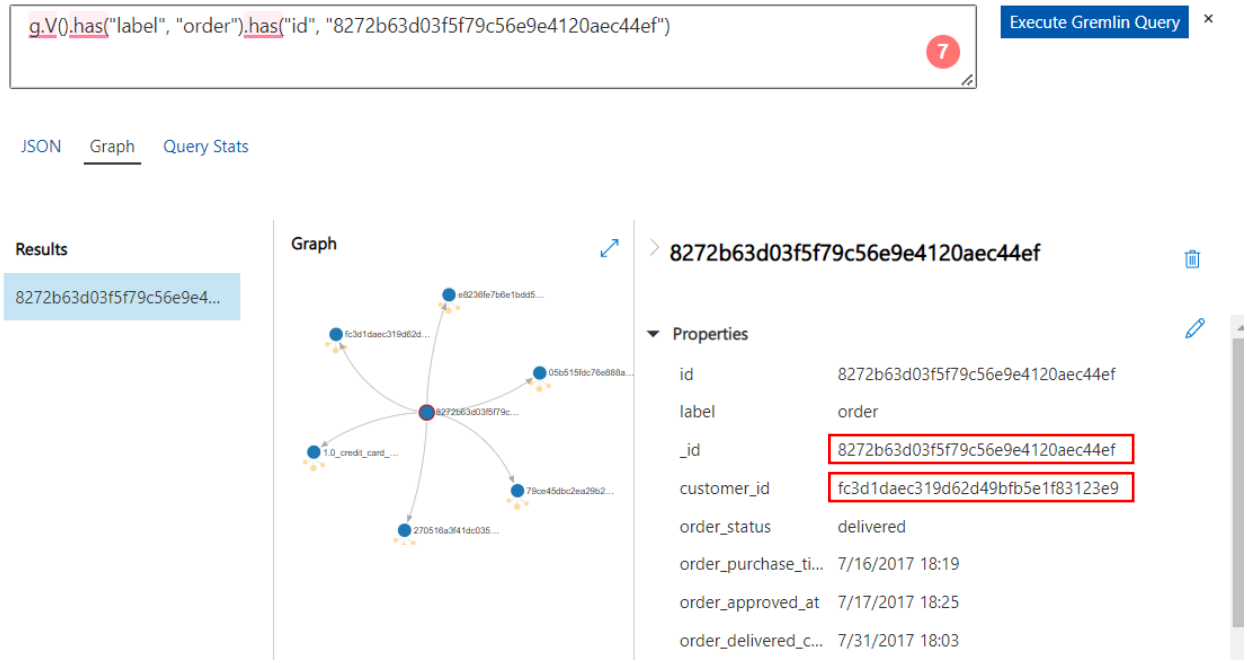
Count of Vertices:



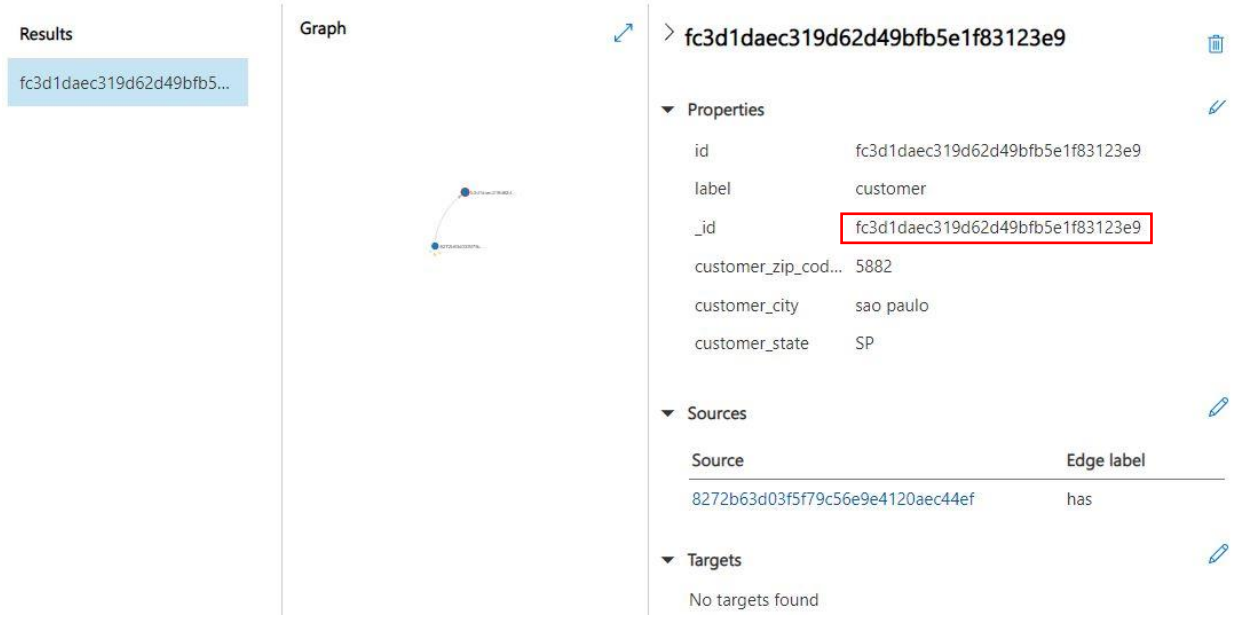
Count of Edges:

Azure Cosmos Graph DB:

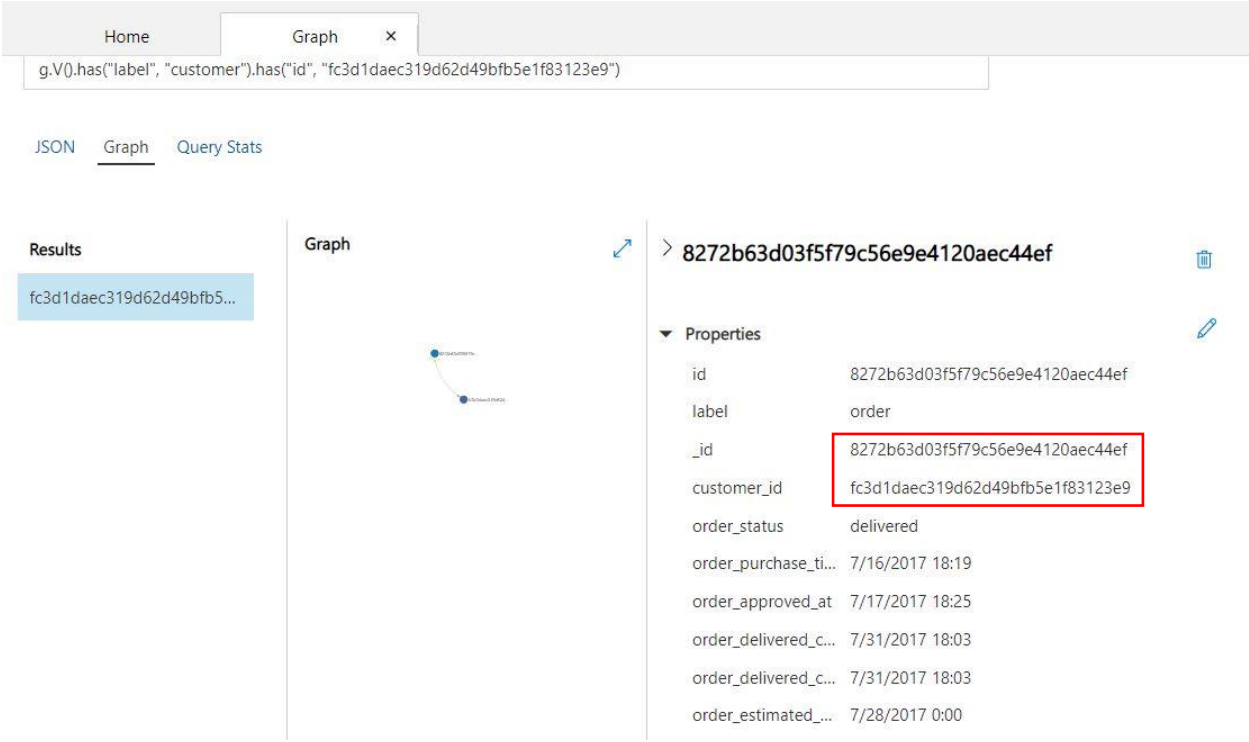
An Order having multiple nodes: Customer, Product Payment, OrderReview



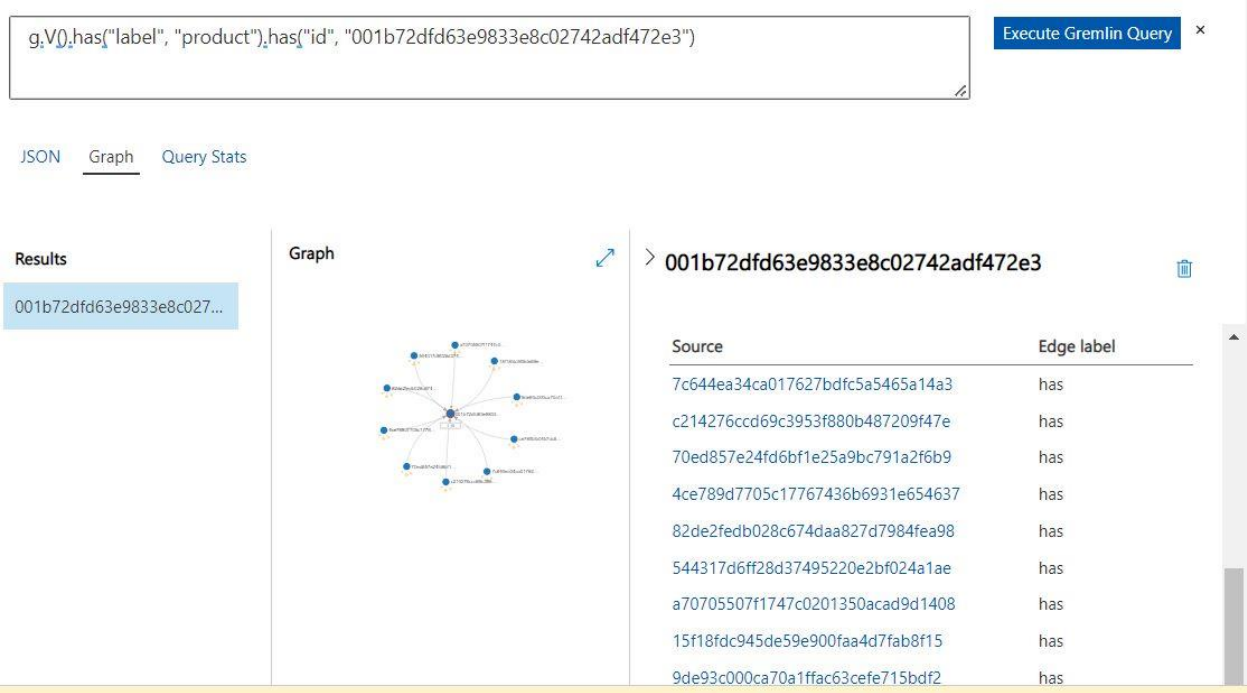
The same Customer:



Same Customer having the Order:



A Product having Multiple Orders:



In this way, we have successfully implemented Document DB and Graph DB using NoSQL API and Gremlin API, respectively.