P-3: Implementation

Brazilian E-commerce Dataset

Team 4:
Anushka Darade: NUID: 002734159
Yash Panchal NUID: 002771456
Dhriti Kanchan: NUID: 002794620
Vikash Singh: NUID: 002929181
Swapnil Bhasgauri NUID: 002752978

Implementation Description:

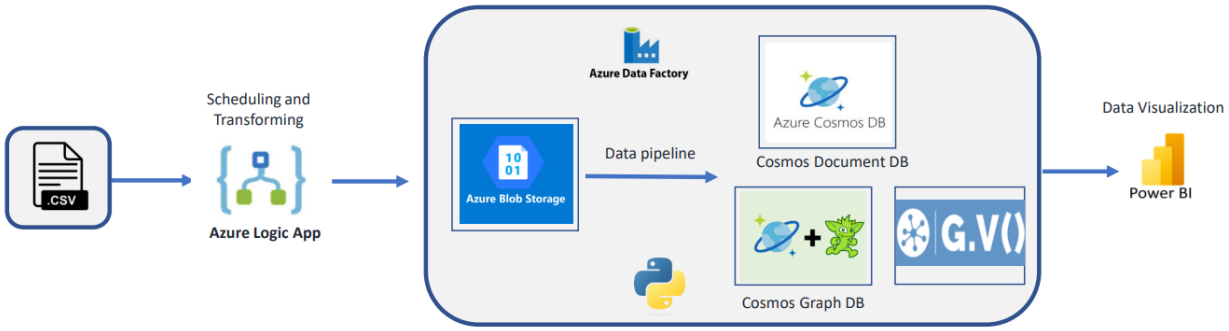
An overview of the implementation procedure is as follows:

1.Upon conducting Data Profiling for the Brazilian E-commerce dataset, we observed and addressed issues such as missing values, inconsistent data, and ensured alignment of data types with respect to the data dictionary. Initially, we created a data dictionary specific to the Brazilian E-commerce dataset, outlining column names and corresponding data types.

2. The files from the blob storage are imported into Azure Data Factory in the Dataset section for the Azure Cosmos Document DB implementation. 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. The first step in implementing Azure Graph DB is to utilize Azure Data Factory to merge the various.csv files needed for the graph implementation into a single file. We created a Python connection to our Azure Blob Storage, which is where the combined file is stored, so that it can be retrieved immediately. Next, using the Gremlin API, created a Python connection to the Azure Cosmos Graph Database. After the connection was made, we used the Gremlin API to generate the vertices and edges and ingested data into Azure Cosmos Graph DB.

Data Architecture Diagram:



Data Catalog:

olist_customers_dataset

customer_id	v_string	
customer_unique_id	v_string	
customer_zip_code_prefix	v_string	
customer_city	v_string	
customer_state	v_string	

olist_geolocation_dataset

geolocation_zip_code_prefix	v_string	
geolocation_lat	v_string	
geolocation_lng	v_string	
geolocation_city	v_string	
geolocation_state	v_string	

olist_order_items_dataset

order_id	v_string	
order_item_id	v_string	
product_id	v_string	
seller_id	v_string	
shipping_limit_date	v_string	
price	v_string	
freight_value	v_string	

olist_order_payments_dataset

order_id	v_string	
payment_sequential	v_string	
payment_type	v_string	
payment_installments	v_string	
payment_value	v_string	

olist_order_reviews_dataset

review_id	v_string	
order_id	v_string	
review_score	v_string	
review_comment_title	v_string	Null (87656)
review_comment_message	v_string	Empty and null (58247) Cleaned Null, blank, white spaces, //////////, *******, … …..
review_creation_date	v_string	Remove the timestamp part from the data
review_answer_timestamp	v_string	Remove the timestamp part from the data

olist_orders_dataset

order_id	v_string	
customer_id	v_string	
order_status	v_string	
order_purchase_timestamp	v_string	
order_approved_at	v_string	Null (160)
order_delivered_carrier_date	v_string	Null (1783)
order_delivered_customer_date	v_string	Null (2965), Remove Timestamp from the data and replace with 1900-01-01
order_estimated_delivery_date	v_string	Remove Timestamp from the data and replace with 1900-01-01

olist_products_dataset

product_id	v_string	
product_category_name	v_string	Null (610), Replacing Null with “No Value Provided”
product_name_lenght	v_string	Null (610), Replacing Null with “No Value Provided”
product_description_lenght	v_string	Null (610), Replacing Null with “No Value Provided”
product_photos_qty	v_string	Null (610), Replacing Null with “-1”
product_weight_g	v_string	Null (2), Replacing Null with “-1”
product_length_cm	v_string	Null (2), Replacing Null with “-1”
product_height_cm	v_string	Null (2), Replacing Null with “-1”
product_width_cm	v_string	Null (2), Replacing Null with “-1”

olist_sellers_dataset

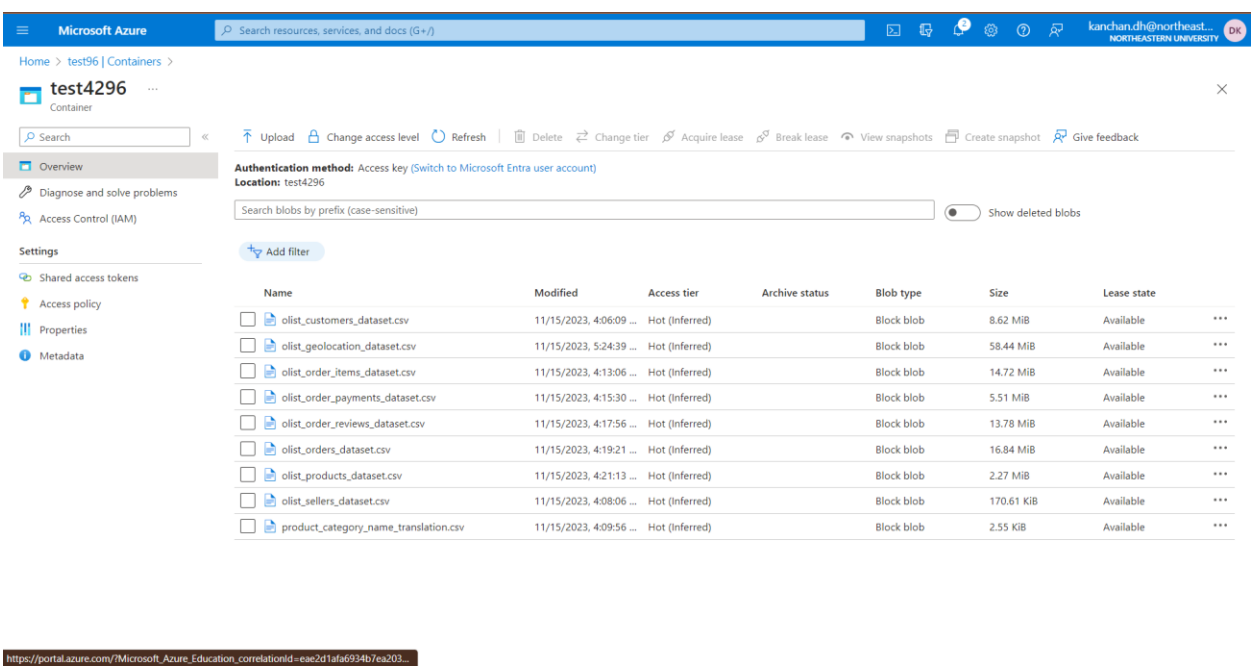
seller_id	v_string	
seller_zip_code_prefix	v_string	
seller_city	v_string	
seller_state	v_string	

product_category_name_translation

product_category_name	v_string	
product_category_name_english	v_string	

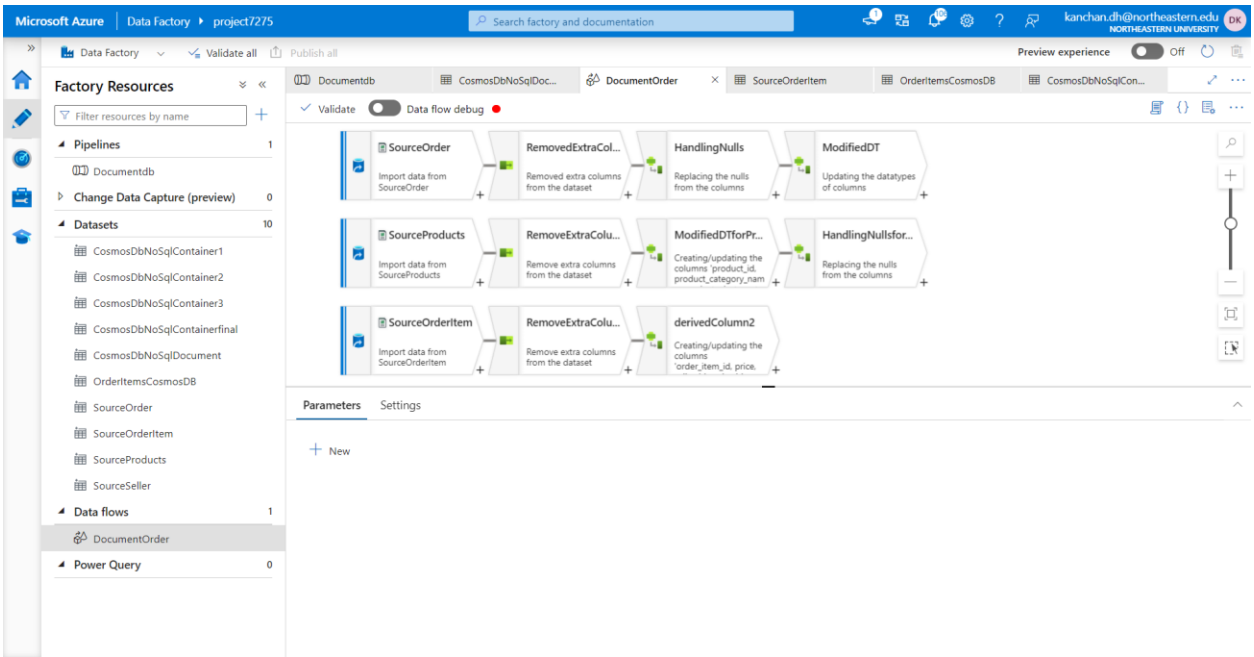
Implementation:

Files in Azure Blob Storage:



AZURE COSMOS DOCUMENT DB implementation:

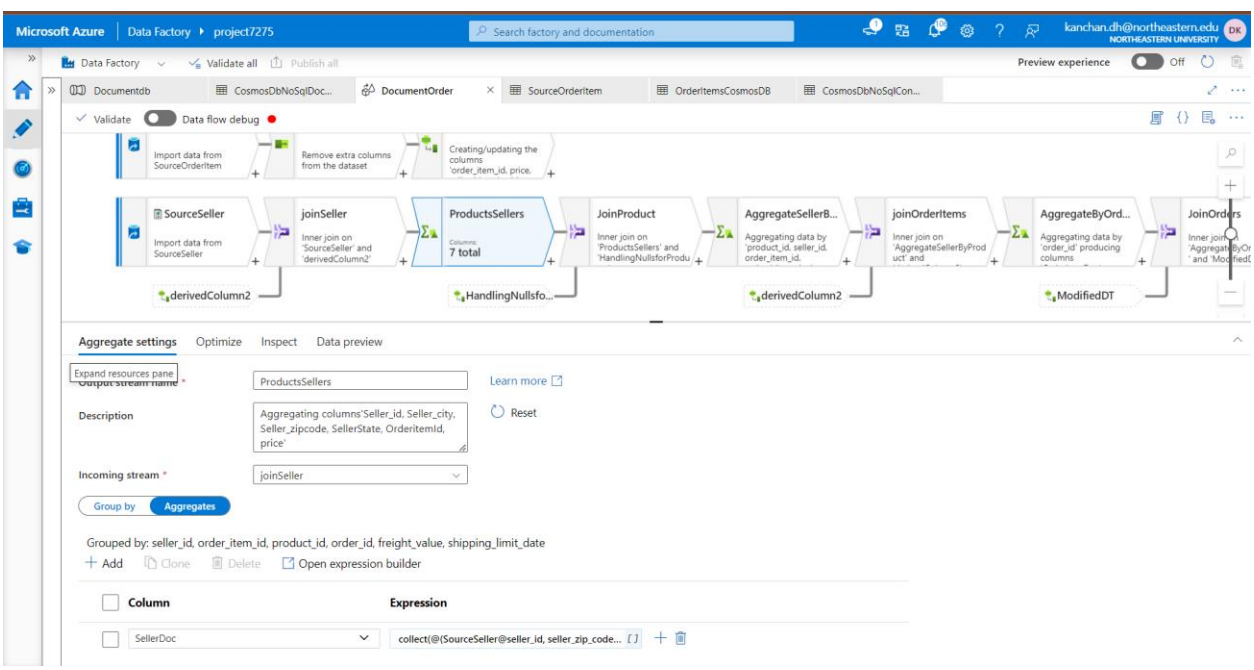
In Azure Data Factory, we created dataflow to carry out various data transformation and cleaning functions on.csv files, such as: deleting unnecessary columns, managing null values in columns, and changing datetime, timestamp, and other data types to the appropriate format and data type.



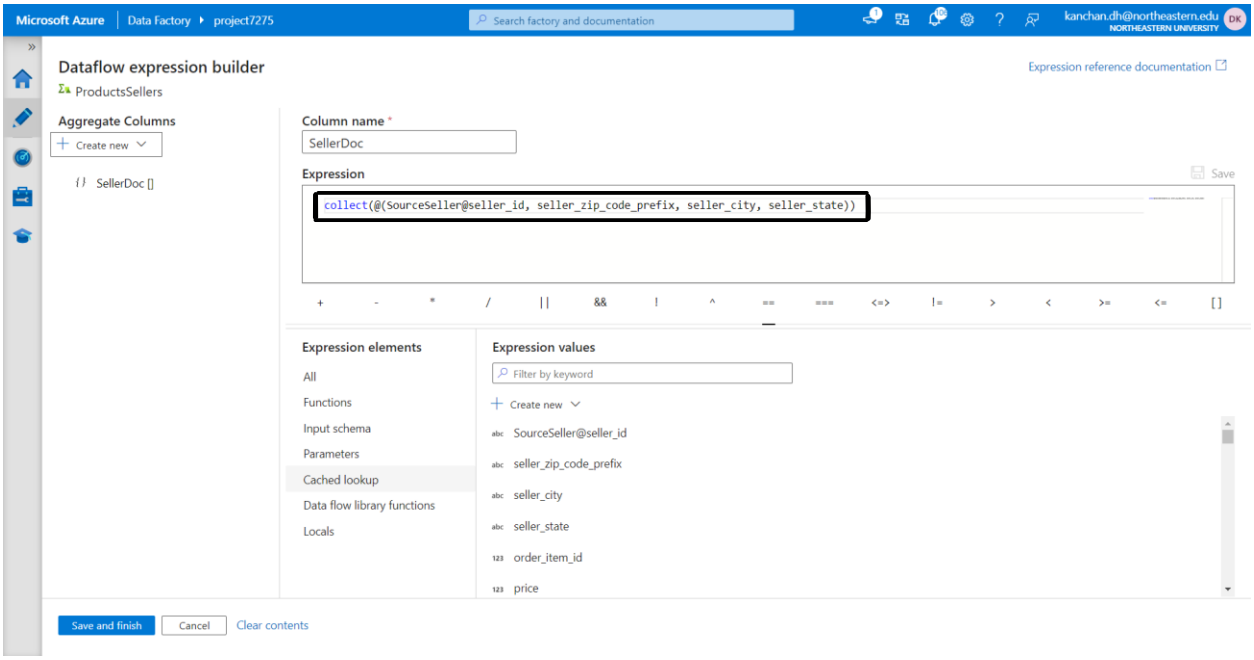
Applying joins to combine data from multiple. csvs.

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

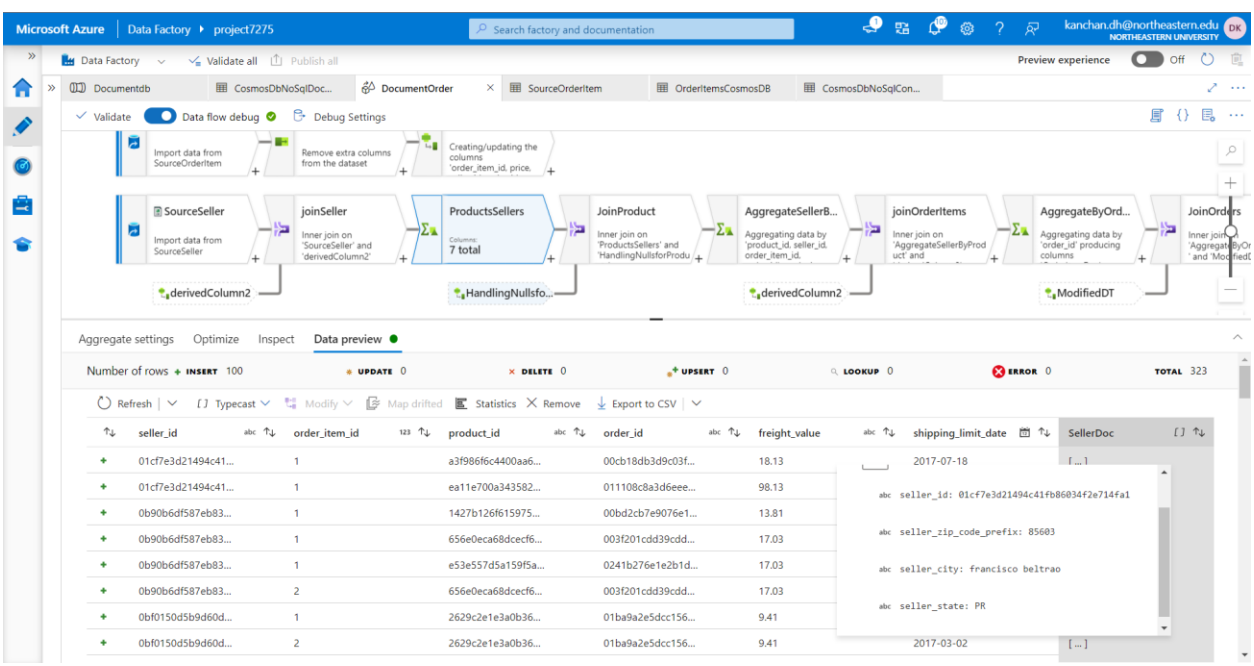
Below, aggregated by seller_id:



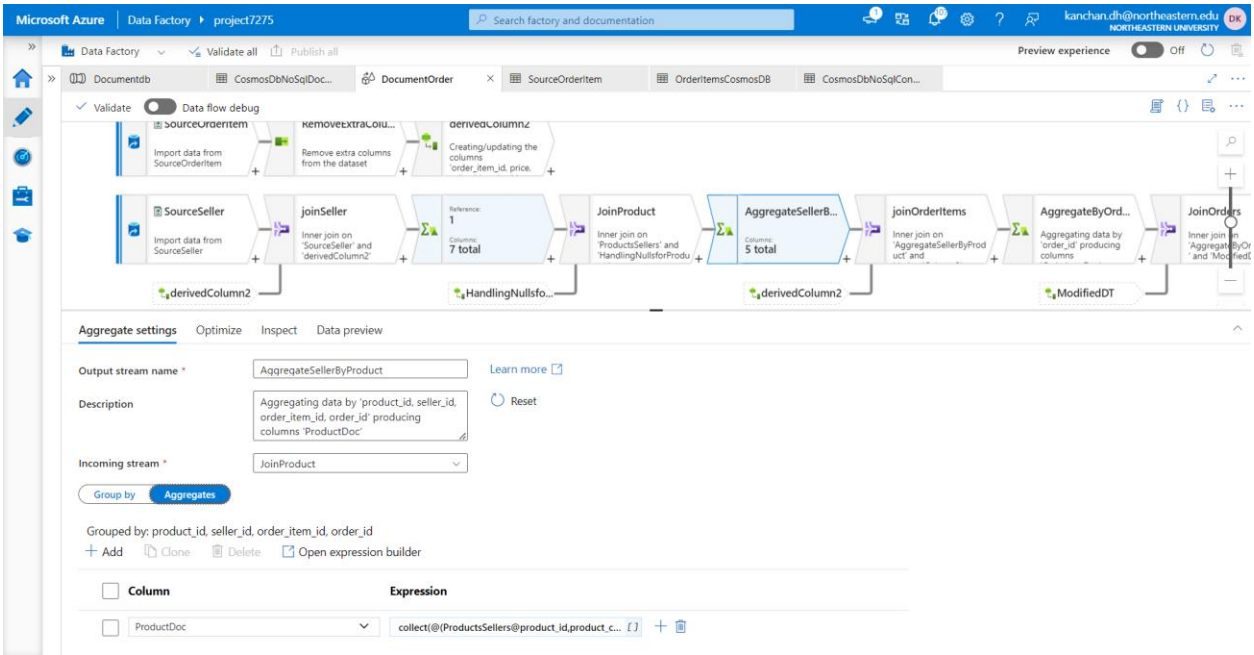
Expression used for Aggregation: collect(@(id, column_1, column_2,….., column_N)

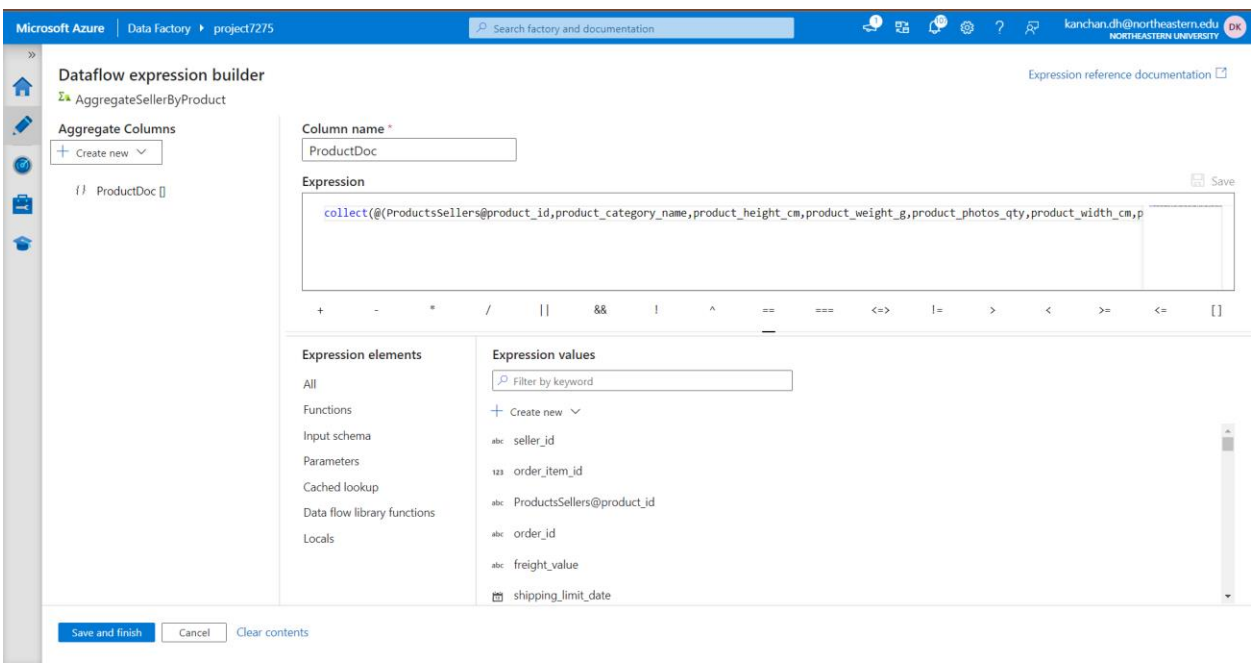


Data Preview:

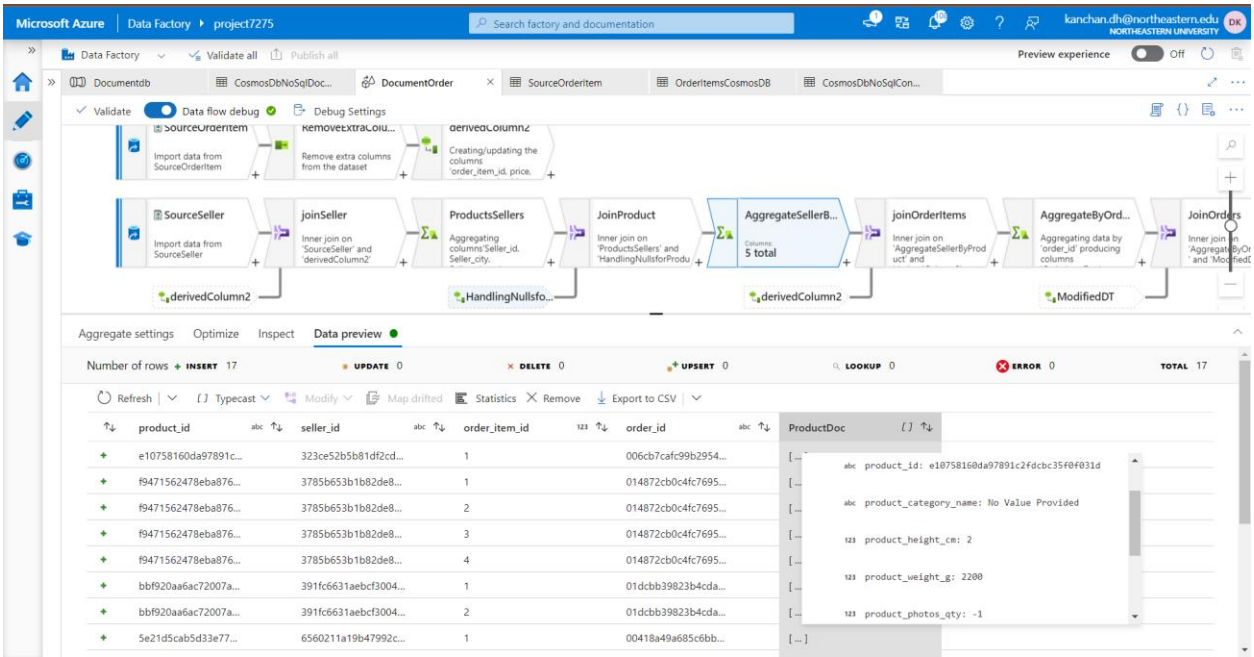


Aggregated by respective ids from the product table to achieve the nested document db json format:

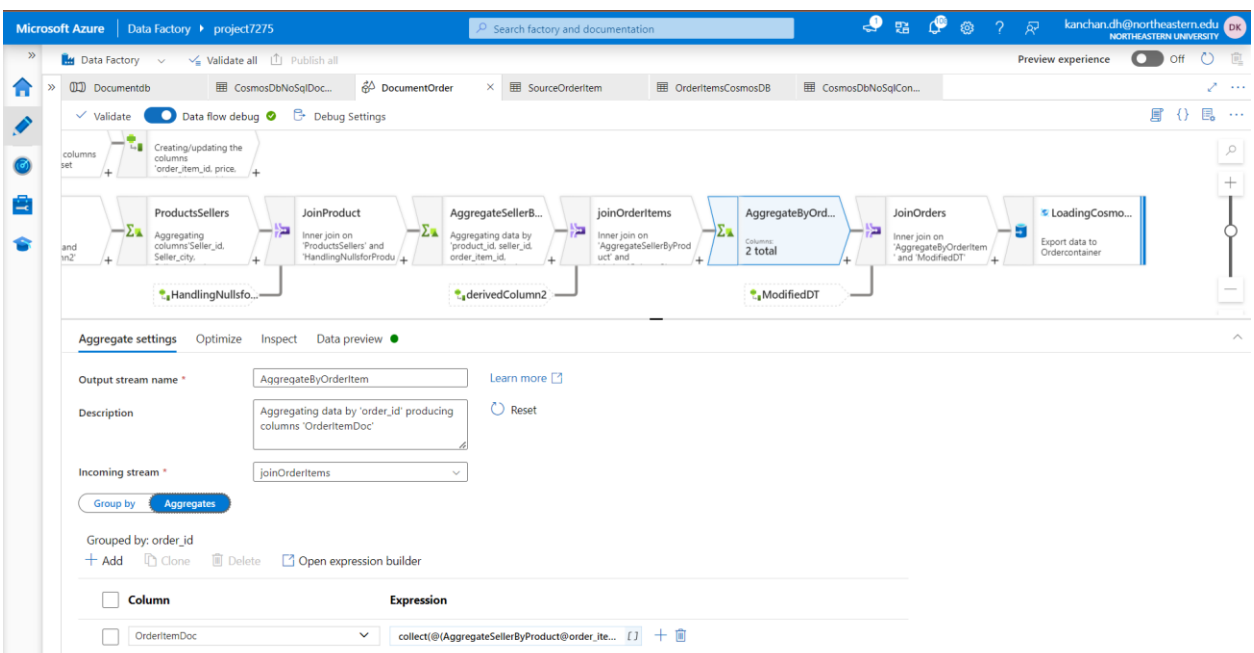


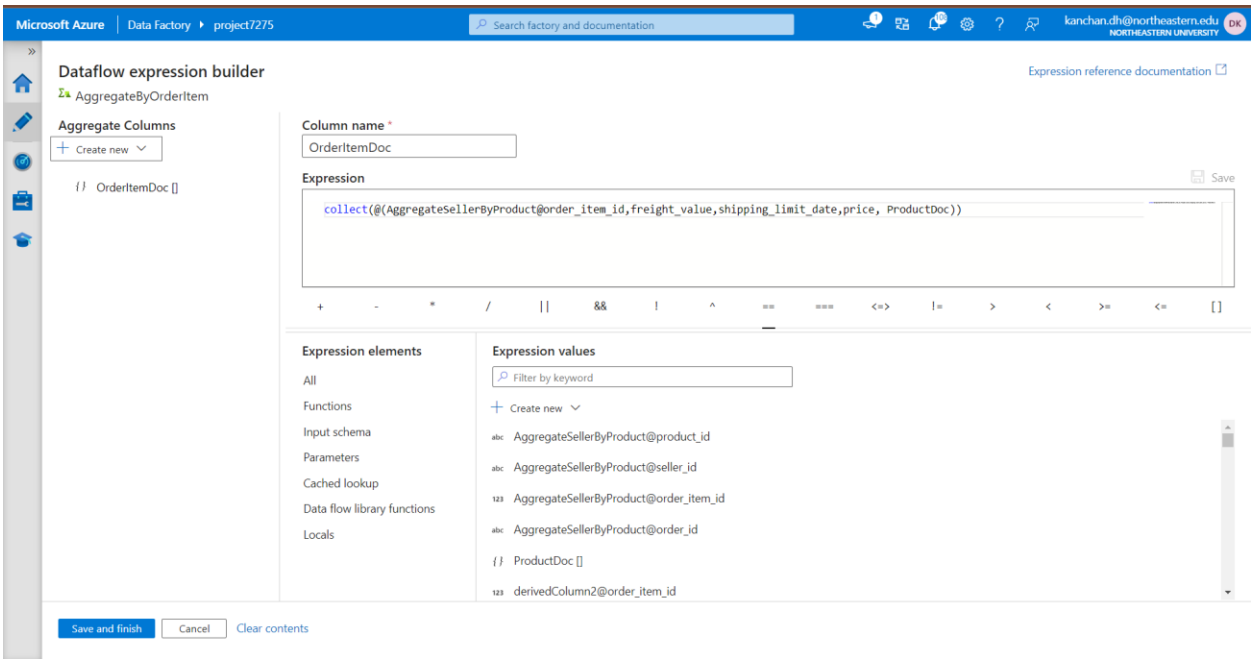


Data Preview:

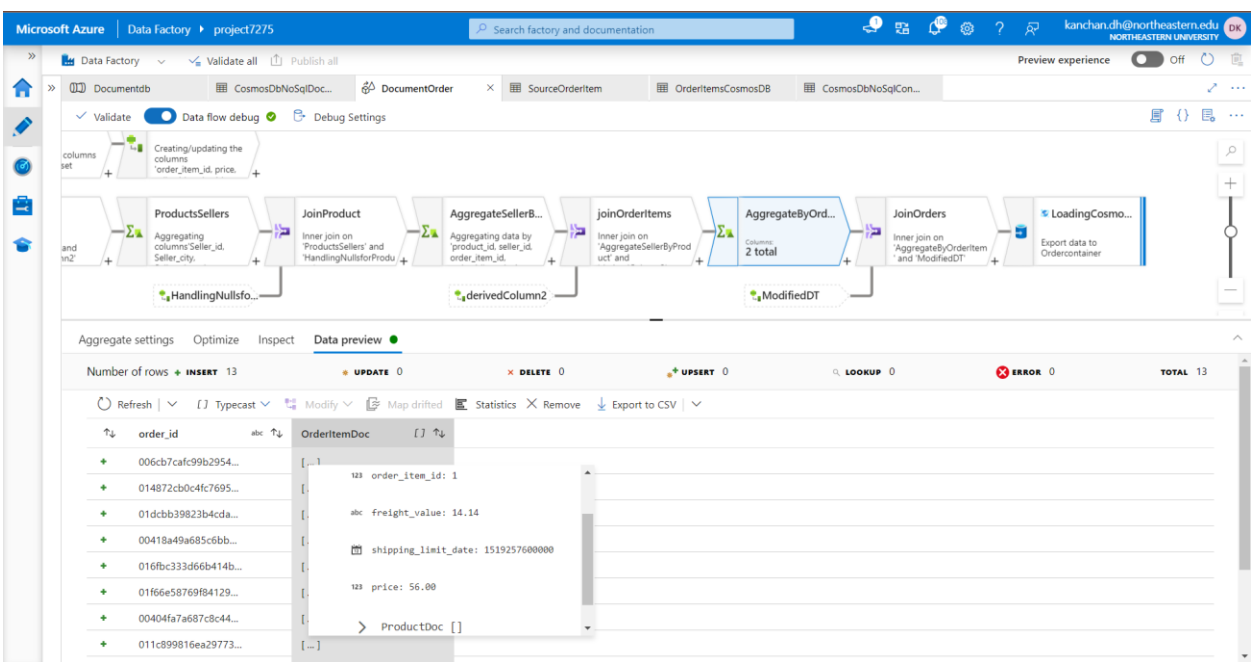


Aggregated by respective ids from the Order Item table to achieve the nested document db json format and creating a column named '**OrderItemDoc**':

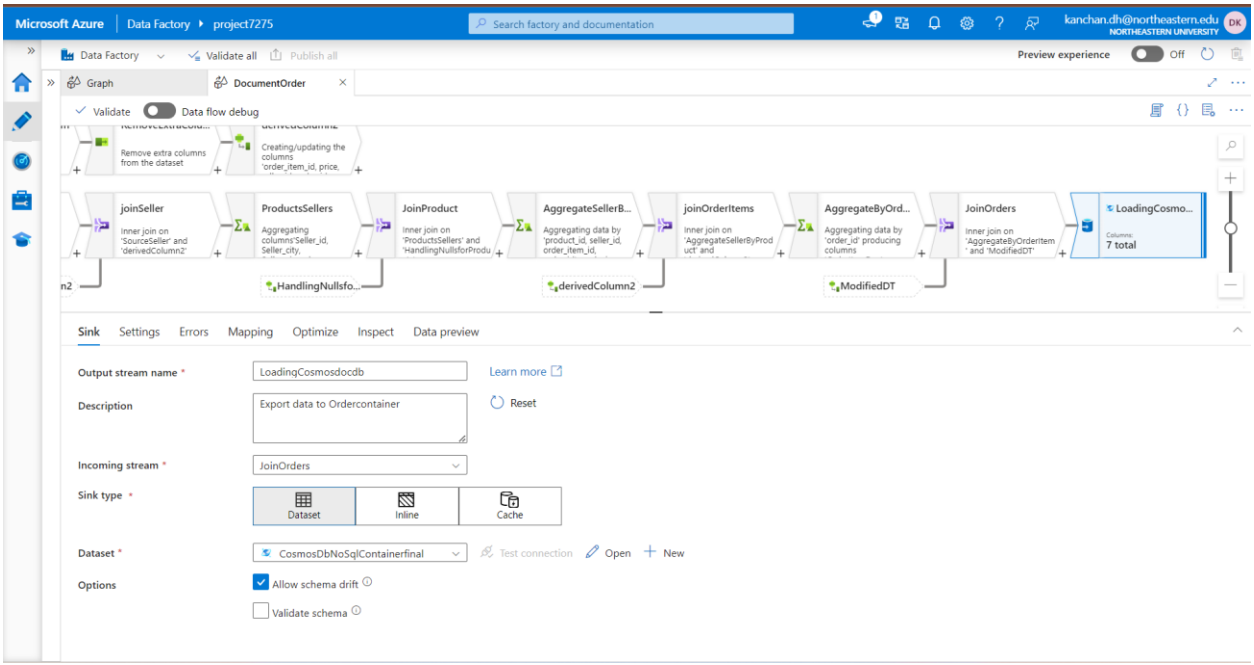




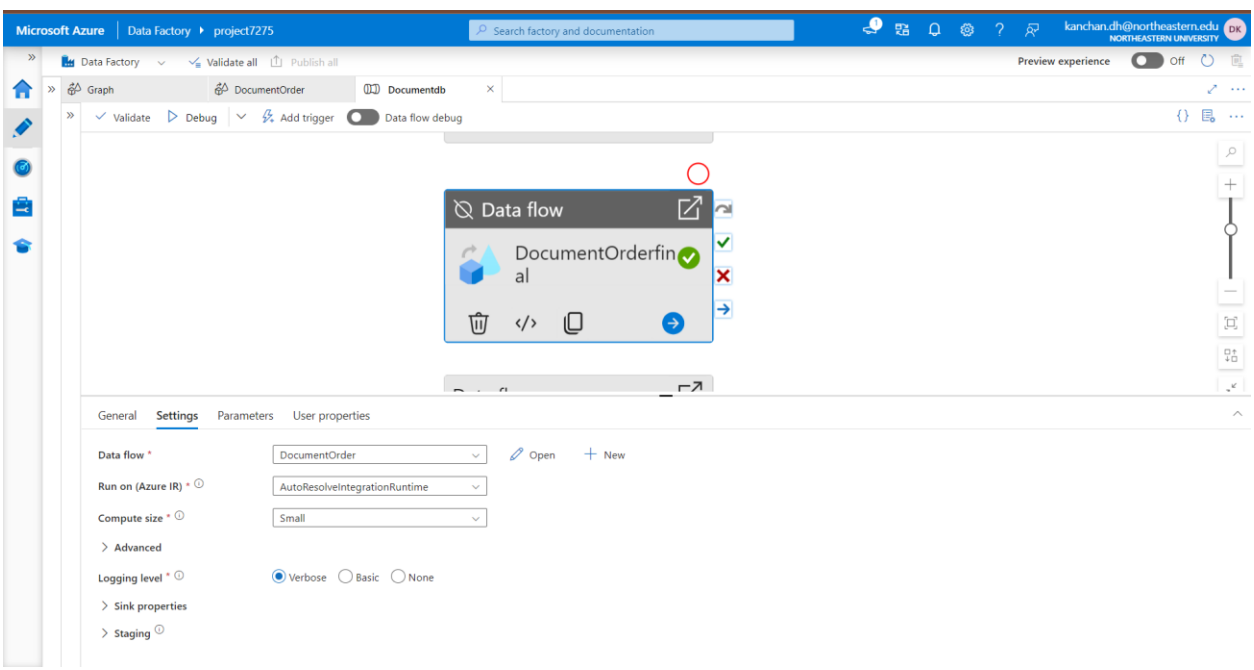
Data Preview:



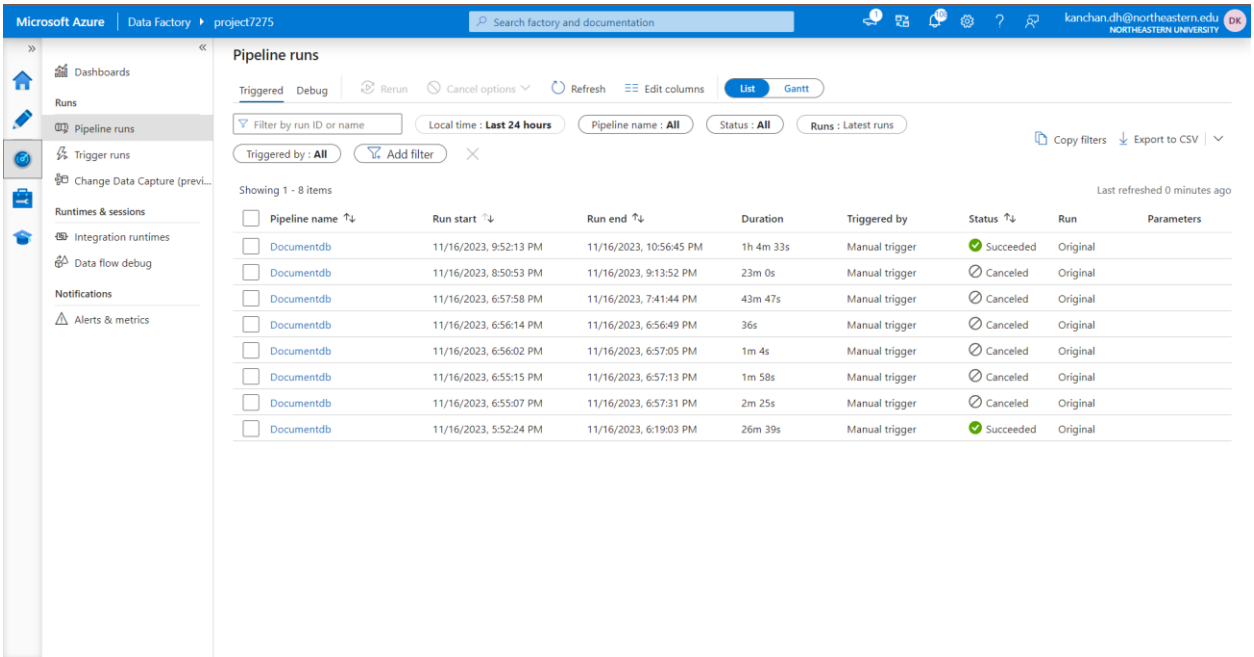
Last step to sink the data into Azure Cosmos Document DB:



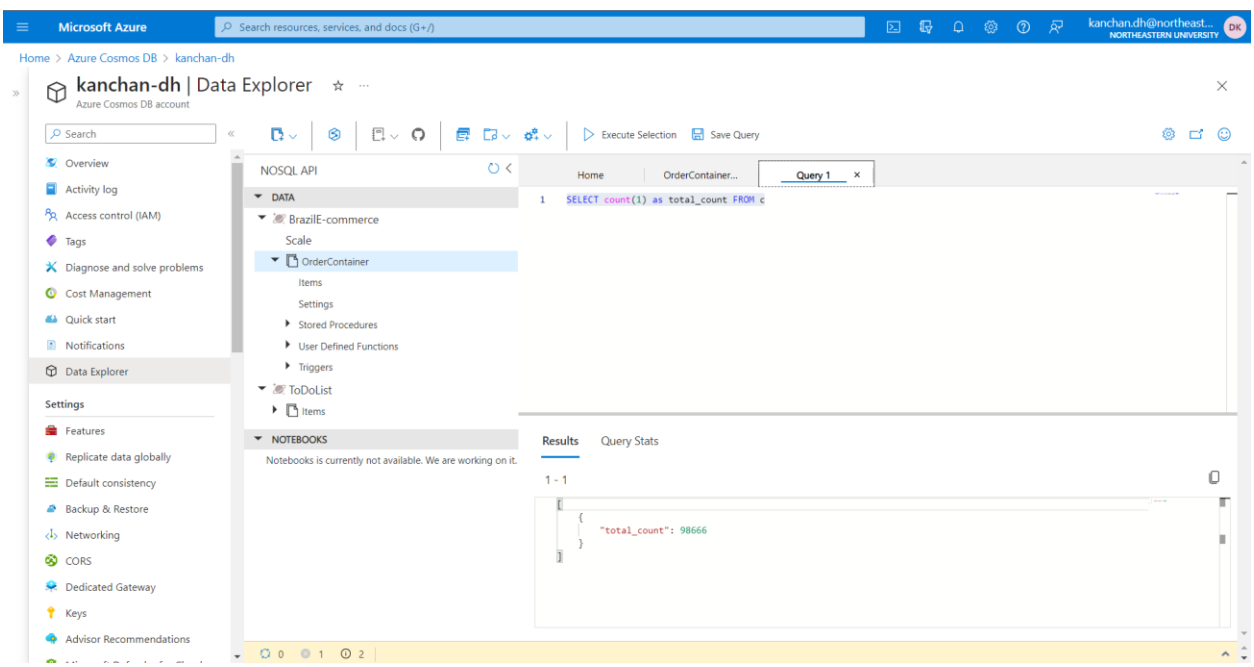
Running the data pipeline:



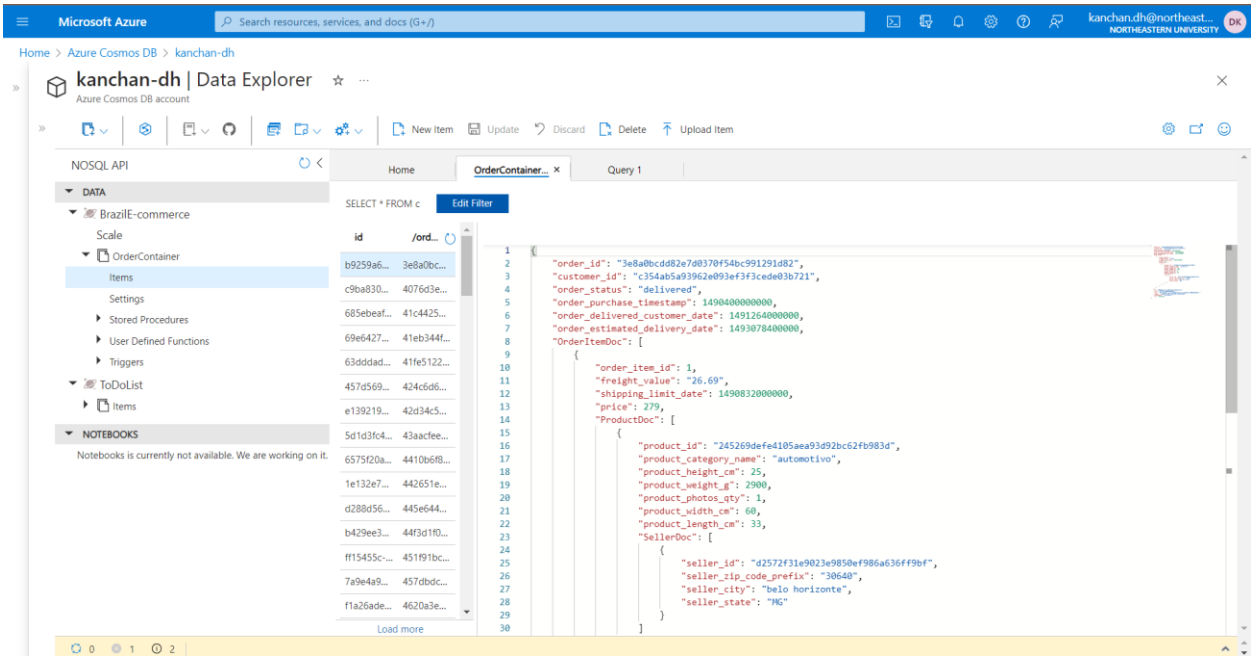
Successful execution of the data pipeline:



Count of Data loaded in Azure Cosmos Document DB:



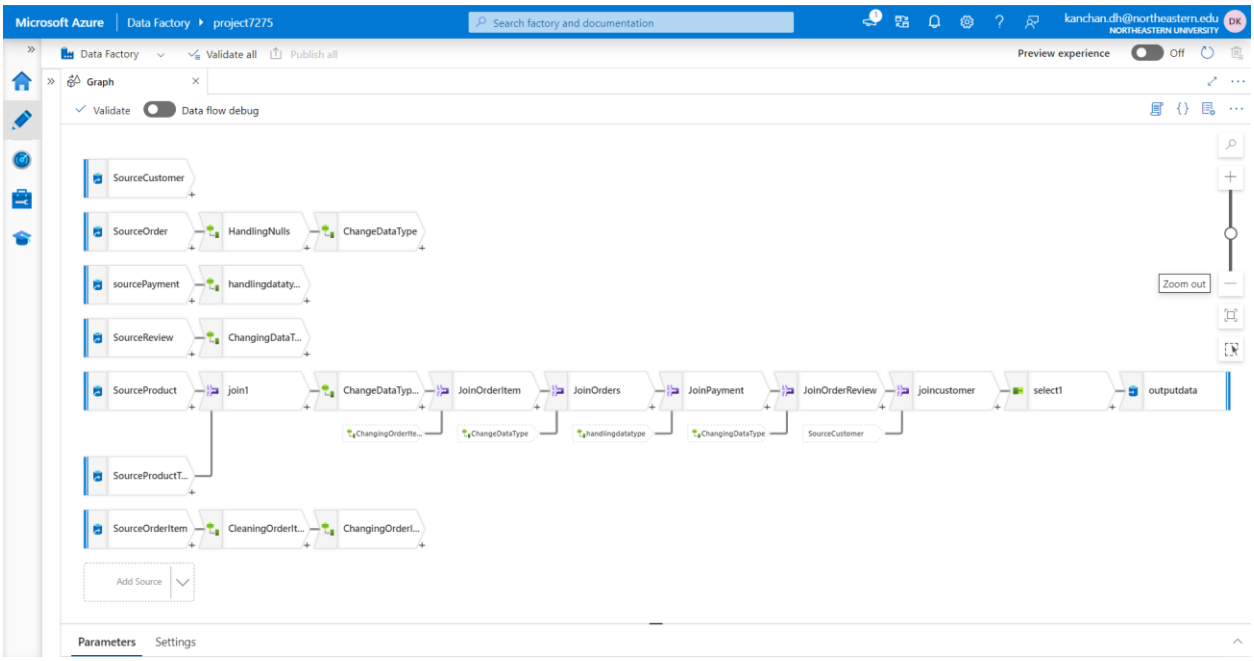
Document DB loaded in Azure:



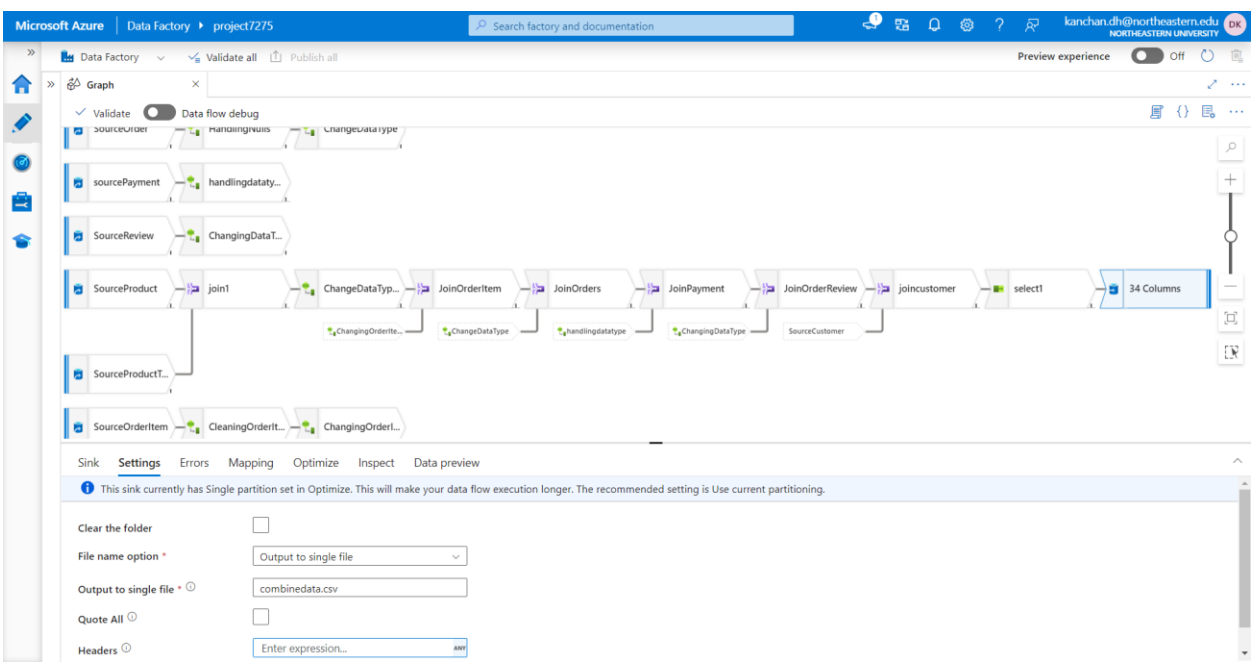
Revised Document DB Data model: (Combined Order, OrderItems, Products, Seller and Geolocation into one Document DB as per querying needs)

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

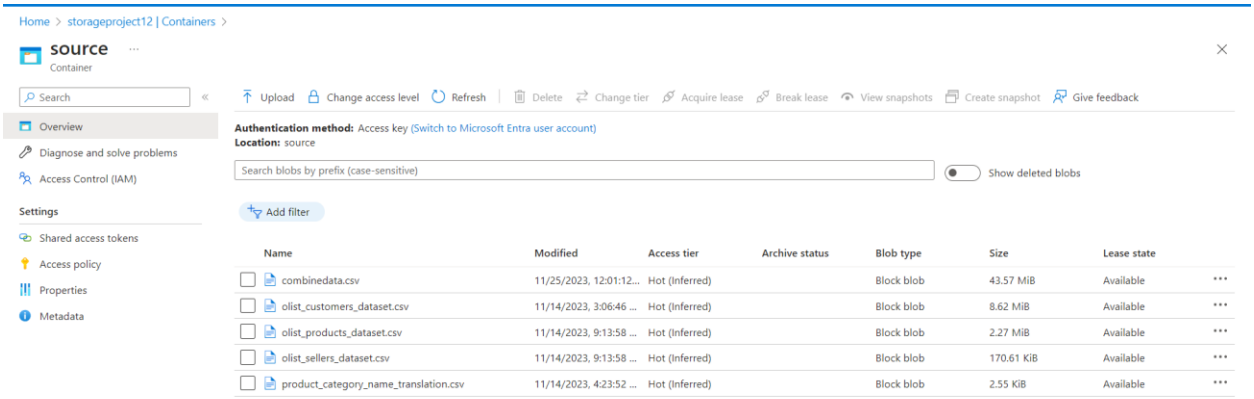
AZURE COSMOS GRAPH DOCUMENT DB implementation using GREMLIN API: Created a Dataflow in Azure Data Factory to perform some data cleansing and transformation processes on .csv files like: Removing Extra Columns, Handling Null Values in columns, Modifying the datetime, timestamp and other data types to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



Generating combinedata.csv file:



Combined .csv file stored in Azure blob storage:



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

```python
def generate_gremlin_code(df, vertex_label: str, pk_column: str): #List[str]
  gremlin_code = []
  for i, row in df.iterrows():
      pk_value = str(row[pk_column])
      properties = []
      for col, value in row.items():
            if col != pk_column:
                value = value.replace("'", r"\'")
                properties.append(f".property('{col}', '{value}')")
      properties = "".join(properties)
      vertex_code = f"g.addV('{vertex_label}').property('seller_id', '1').property('id', '{pk_value}').property('_id', '{pk_value}'){properties}"
      gremlin_code.append(vertex_code)
  return gremlin_code
```

Python

Function for inserting Vertices:

```python
warnings.filterwarnings('ignore')
def insert_vertices(client, all_vertices):
  for idx, query in enumerate(all_vertices):
      if idx % 800 == 0:
          time.sleep(3)
      try:
          client.submit(query)
      except protocol.GremlinServerError as e:
          print("Gremlin Server Error:", e.status_code)
      except Exception as e:
          print("An unexpected error occurred:===> ", query)
```

Python

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

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

```python
def create_gremlin_query(df, col1, col2):
  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
```

Python

Function for inserting Edges:

```python
def insert_edges(client, all_edges):
    for idx, query in enumerate(all_edges):
        if idx % 800 == 0:
            time.sleep(3)
        try:
            client.submit(query)
        except protocol.GremlinServerError as e:
            print("Gremlin Server Error:", e.status_code)
        except Exception as e:
            print("An unexpected error occurred:===> ", query)
```

Python

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

Establishing connection with the azure cosmos db gremlin database(specifed database)

```python
def getClient():
    gremlin_client = client.Client(cosmos_db_endpoint, 'g',
                                   username=f"/dbs/{database_name}/colls/{graph_name}",
                                   password=cosmos_db_password,
                                   message_serializer=serializer.GraphSONSerializersV2d0())
    return gremlin_client
```

Python

Creating Dataframe for vertices and dropping duplicate values in the columns

```python
order = dataFrame[['order_id','order_estimated_delivery_date', 'order_delivered_carrier_date', 'order_approved_at', 'order_purchase_timestamp', 'order_status']].copy()
print(f'Ord Loading... unique check: {len(order)}')
order = order.drop_duplicates(subset=['order_id'])
print(f'Order after unique check: {len(order)}')

product = dataFrame[['product_id', 'product_height_cm', 'product_length_cm', 'product_weight_g', 'product_photos_qty', 'product_category_name', 'product_width_cm']].copy()
print(f'Product before unique check: {len(product)}')
product = product.drop_duplicates(subset=['product_id'])
print(f'Product after unique check: {len(product)}')

review = dataFrame[['review_id', 'review_creation_date', 'review_score', 'review_answer_timestamp']].copy()
print(f"Review before unique check full-data: {len(review)}")
print(f"Review before unique check: {len(review['review_id'].unique())}")
review = review.drop_duplicates(subset=['review_id'])
print(f"Review before unique check full-data: {len(review)}")
print(f"Review before unique check: {len(review['review_id'].unique())}")

customer = dataFrame[['customer_id', 'customer_city', 'customer_state']].copy()
print(f"Customer before unique check full-data: {len(customer)}")
print(f"Customer before unique check: {len(customer['customer_id'].unique())}")
customer = customer.drop_duplicates(subset=['customer_id'])
print(f'Customer after unique check full-data: {len(customer)}')
print(f"Customer after unique check: {len(customer['customer_id'].unique())}")

payment = dataFrame[['order_id', 'payment_value', 'payment_sequential', 'payment_type', 'payment_installments']].copy()
payment['p_order_id'] = payment['order_id'] + payment['payment_sequential']
print(f"Payment before unique check full-data: {len(payment)}")
print(f"Payment before unique check: {len(payment['p_order_id'].unique())}")
payment = payment.drop_duplicates(subset=['p_order_id'])
print(f'Payment after unique check full-data: {len(payment)}')
print(f"Payment after unique check: {len(payment['p_order_id'].unique())}")
```

Python

Output:

```python
payment = payment.drop_duplicates(subset=['p_order_id'])
print(f'Payment after unique check full-data: {len(payment)}')
print(f"Payment after unique check: {len(payment['p_order_id'].unique())}")
```

Python

```
Order before unique check: 115609
Order after unique check: 96516
Product before unique check: 115609
Product after unique check: 32171
Review before unique check full-data: 115609
Review before unique check: 96319
Review before unique check full-data: 96319
Review before unique check: 96319
Customer before unique check full-data: 115609
Customer before unique check: 96516
Customer after unique check full-data: 96516
Customer after unique check: 96516
Payment before unique check full-data: 115609
Payment before unique check: 100792
Payment after unique check full-data: 100792
Payment after unique check: 100792
```

creating dataframe for edge and dropping duplicate values

```python
order_product = dataFrame[['order_id','product_id']].copy()
print(f'Order Product before unique check: {len(order_product)}')
order_product = order_product.drop_duplicates()
print(f'Order Product after unique check: {len(order_product)}')

order_review = dataFrame[['review_id', 'order_id']].copy()
print(f"Order - Review before unique check full-data: {len(order_review)}")
order_review = order_review.drop_duplicates()
print(f"Order - Review before unique check full-data: {len(order_review)}")

order_customer = dataFrame[['customer_id', 'order_id']].copy()
print(f"Order - Customer before unique check full-data: {len(order_customer)}")
order_customer = order_customer.drop_duplicates()
print(f'Order - Customer after unique check full-data: {len(order_customer)}')

order_payment = payment[['order_id', 'p_order_id']].copy()
print(f"Order - Payment before unique check full-data: {len(order_payment)}")
order_payment = order_payment.drop_duplicates()
print(f'Order - Payment after unique check full-data: {len(order_payment)}')
```

Python

```
Order Product before unique check: 115609
Order Product after unique check: 100157
Order - Review before unique check full-data: 115609
Order - Review before unique check full-data: 97055
Order - Customer before unique check full-data: 115609
Order - Customer after unique check full-data: 96516
Order - Payment before unique check full-data: 100792
Order - Payment after unique check full-data: 100792
```

Inserting vertices in Azure cosmos db

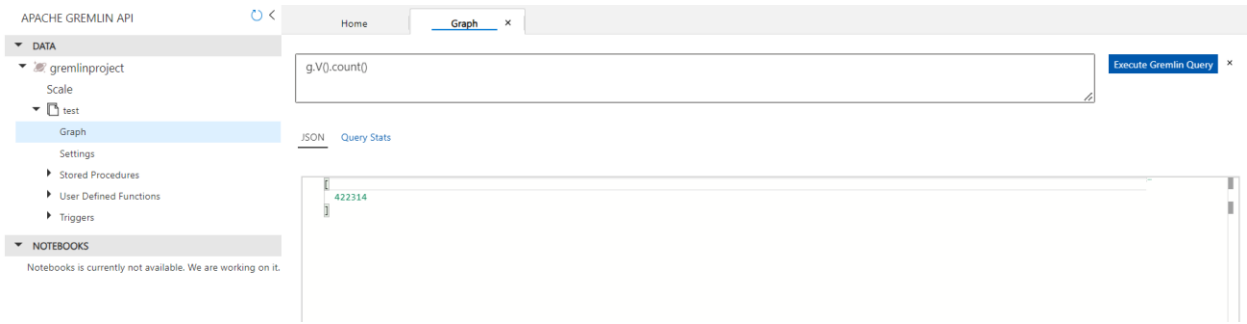
```python
vertex_dfs = [{
    "df": order,
    "v_label": 'order',
    "v_key": 'order_id'
    },
            {
    "df": customer,
    "v_label": 'customer',
    "v_key": 'customer_id'
},
            {
    "df": product,
    "v_label": 'product',
    "v_key": 'product_id'
},
            {
    "df": review,
    "v_label": 'review',
    "v_key": 'review_id'
},
            {
    "df": payment,
    "v_label": 'payment',
    "v_key": 'p_order_id'
}]

def vertex_worker(df, g_client, v_label, v_key):
    insert_vertices(g_client, generate_gremlin_code(df, v_label, v_key))

threads = []
for v_info in vertex_dfs:
    gremlin_client = getClient()
    t = threading.Thread(target=vertex_worker, args=(v_info["df"], gremlin_client, v_info["v_label"], v_info["v_key"]))
    threads.append(t)
    t.start()
```

Insering edges in Azure Cosmos DB

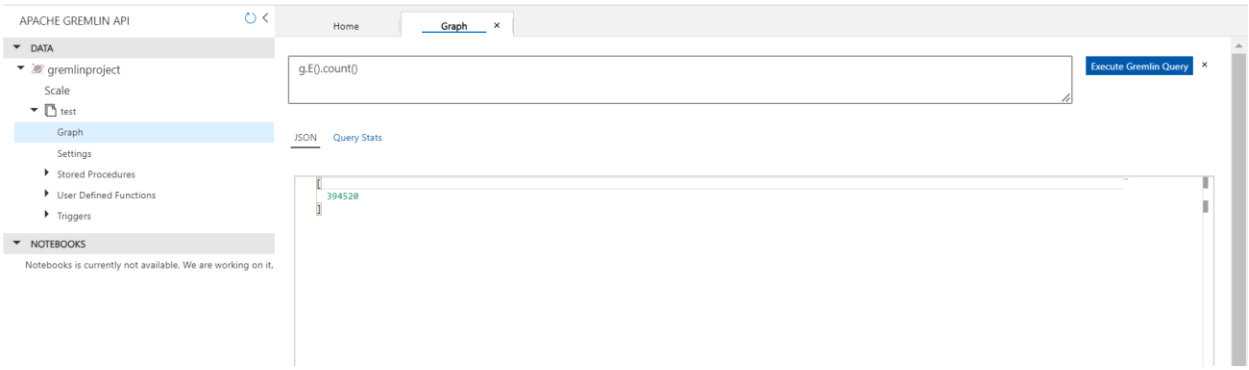
```python
edges_dfs = [{
    "df": order_payment,
    "col1": 'order_id',
    "col2": 'p_order_id'
},
            {
    "df": order_product,
    "col1": 'order_id',
    "col2": 'product_id'
},
            {
    "df": order_review,
    "col1": 'order_id',
    "col2": 'review_id'
},
            {
    "df": order_customer,
    "col1": 'order_id',
    "col2": 'customer_id'
}]

def edge_worker(df, g_client, col1, col2):
    insert_edges(g_client, create_gremlin_query(df, col1, col2))

threads = []
for e_info in edges_dfs:
    gremlin_client = getClient()
    t = threading.Thread(target=edge_worker, args=(e_info["df"], gremlin_client, e_info["col1"], e_info["col2"]))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

Python

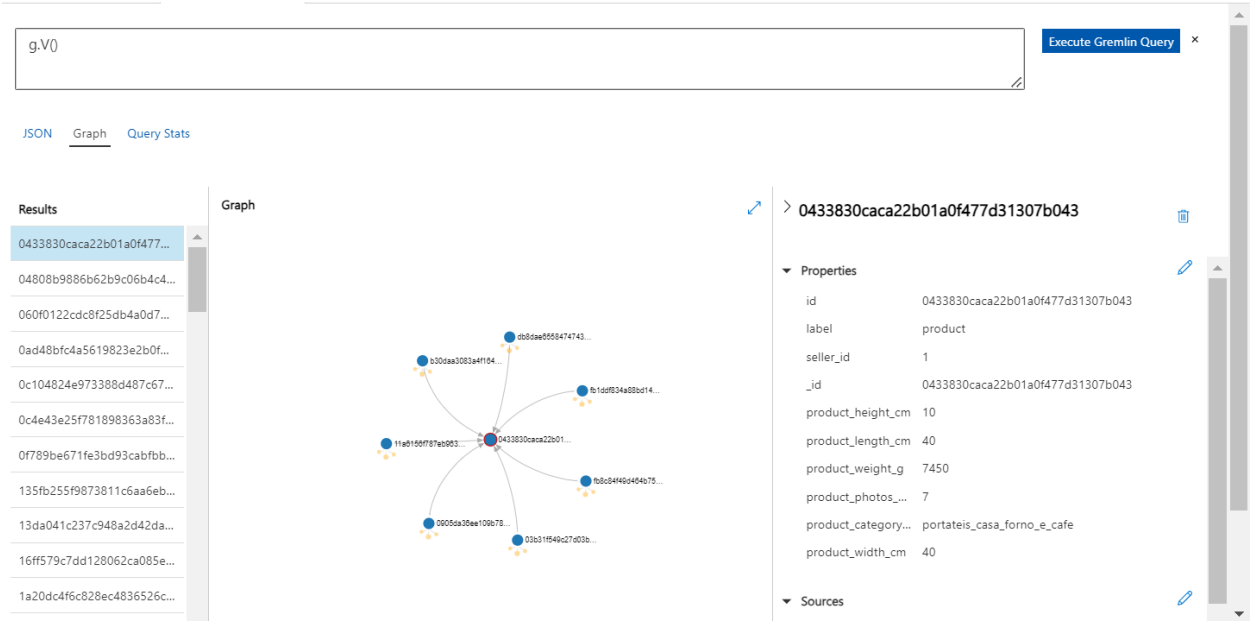
Count of Vertices:

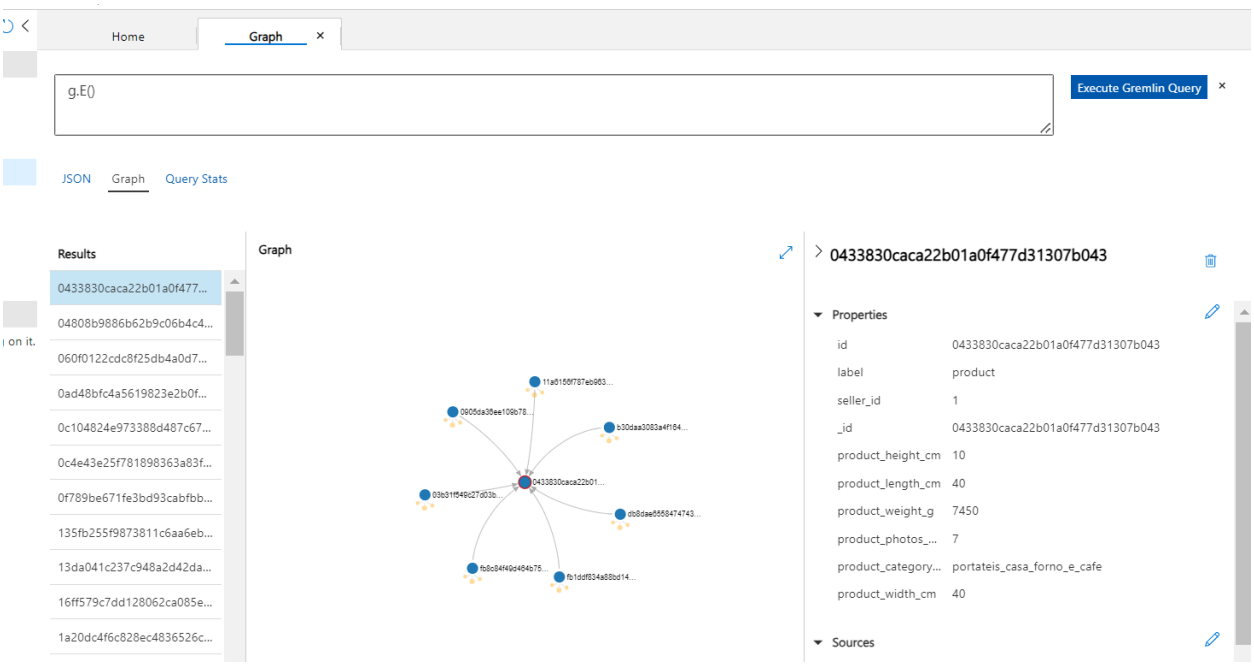


Count of Edges:



Plotting of graphs:





We have successfully implemented Document DB and Graph DB using NoSQL API and Gremlin API, respectively.