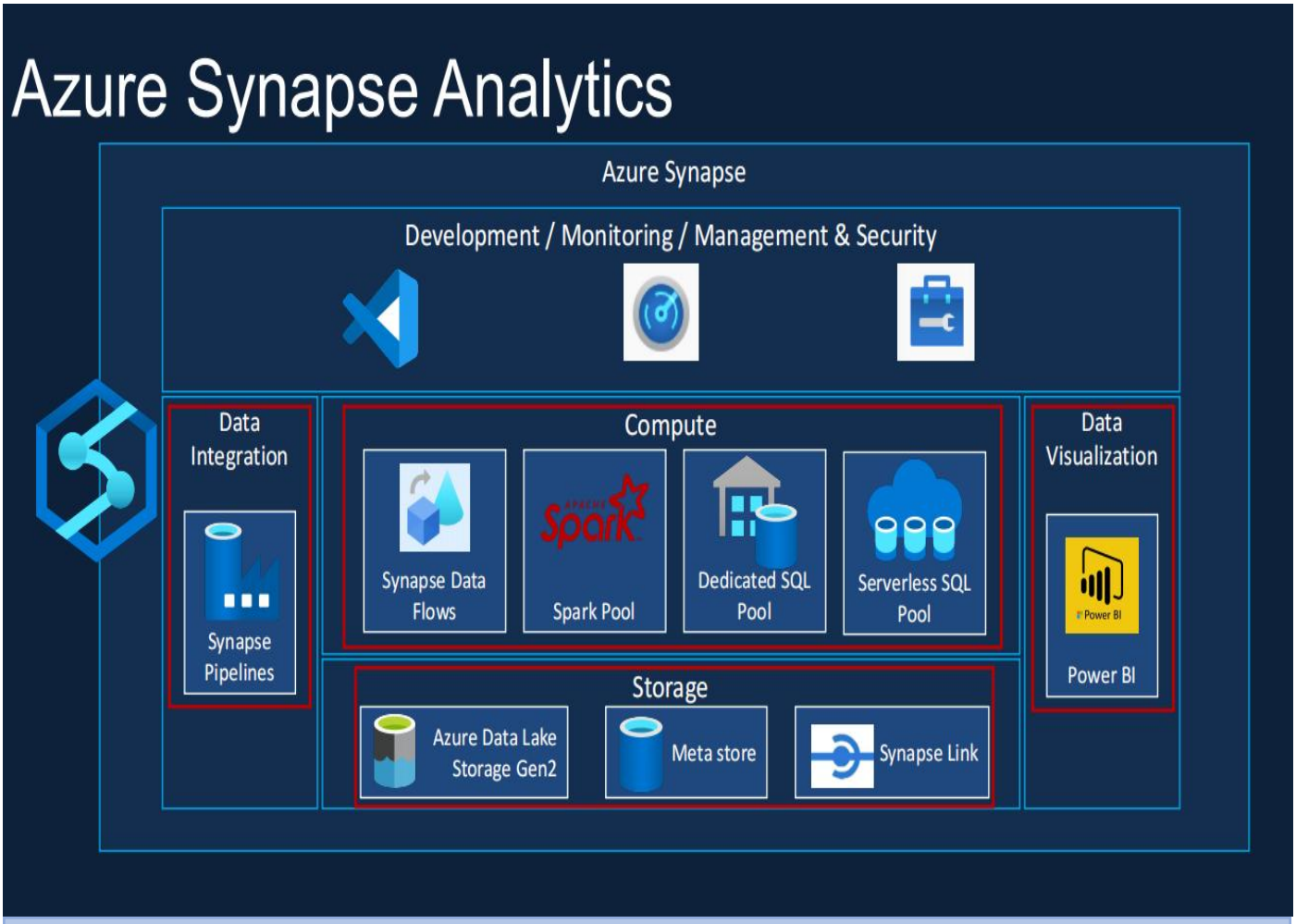


Azure Data Pipeline for COVID-19 Reporting and Analysis

By: Radwa Esmaiel

Azure Synapse Analytics

Overview:

This project focuses on building a scalable and high-performance data engineering solution using Azure Synapse Analytics to analyze and report on New York City Taxi Trips data. The goal is to provide insights into taxi demand, payment behaviors, and other key metrics to support decision-making for the NYC Taxi and Limousine Commission.

The solution leverages the Azure ecosystem to seamlessly process, store, and analyze large datasets while providing actionable insights through Power BI visualizations.

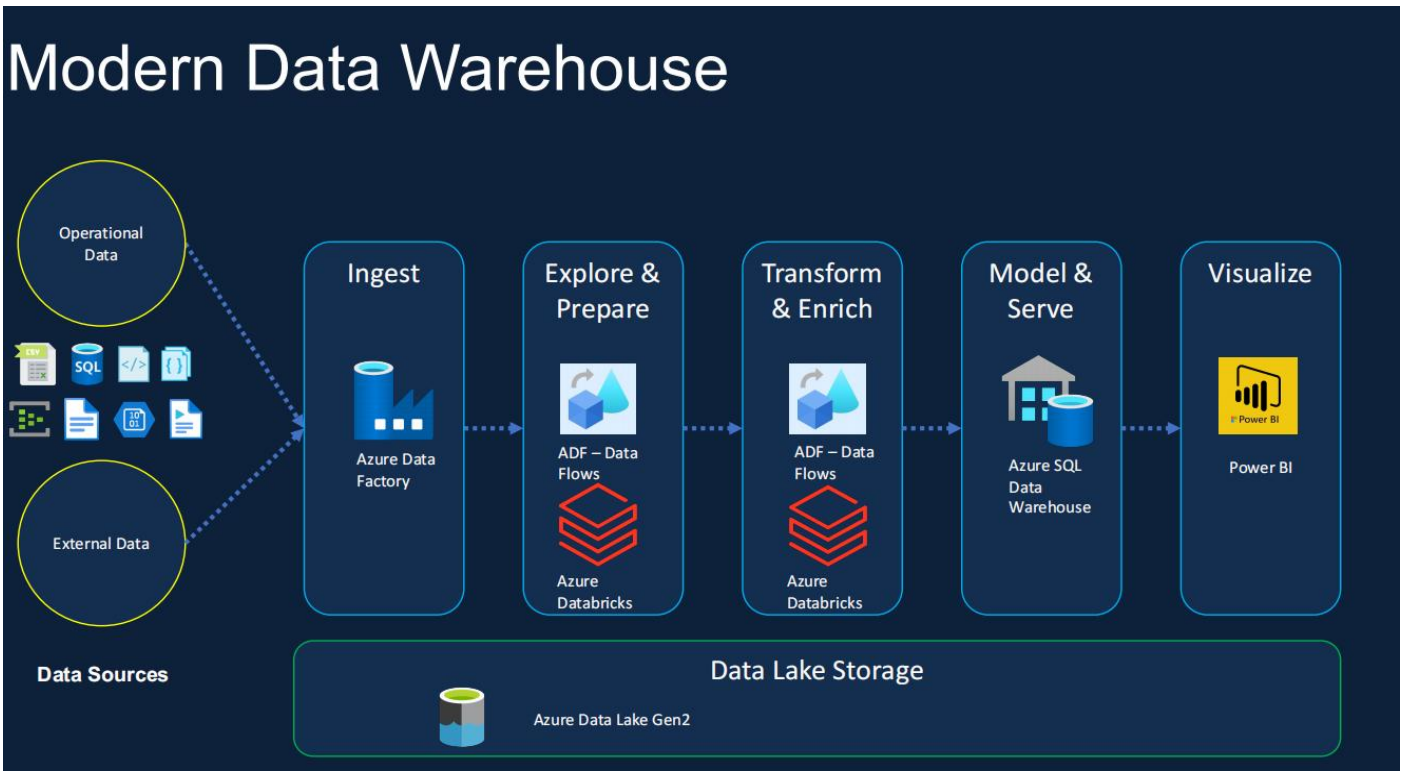
Objectives:

1. Process and analyze large datasets from NYC Taxi Trips.
2. Provide insights on taxi demand, payment behaviors, and borough-specific trends.
3. Create a seamless data pipeline for integrating, transforming, and storing data efficiently.
4. Visualize key metrics using Power BI for decision-making and campaign strategies.

Technologies Used:

- **Azure Synapse Analytics (Server-less SQL Pool, Spark Pool, Dedicated SQL Pool) for distributed data processing and querying.**
- **Azure Data Lake Storage Gen2 for scalable data storage.**
- **Synapse Pipelines for automating the ETL process.**
- **Power BI for creating visualizations and reports.**

This project demonstrates how to design and implement a cloud-based data engineering solution for large-scale data processing, enabling efficient decision-making based on actionable insights from the NYC taxi data.



Step 1: Set Up Azure Synapse Workspace

Configure:

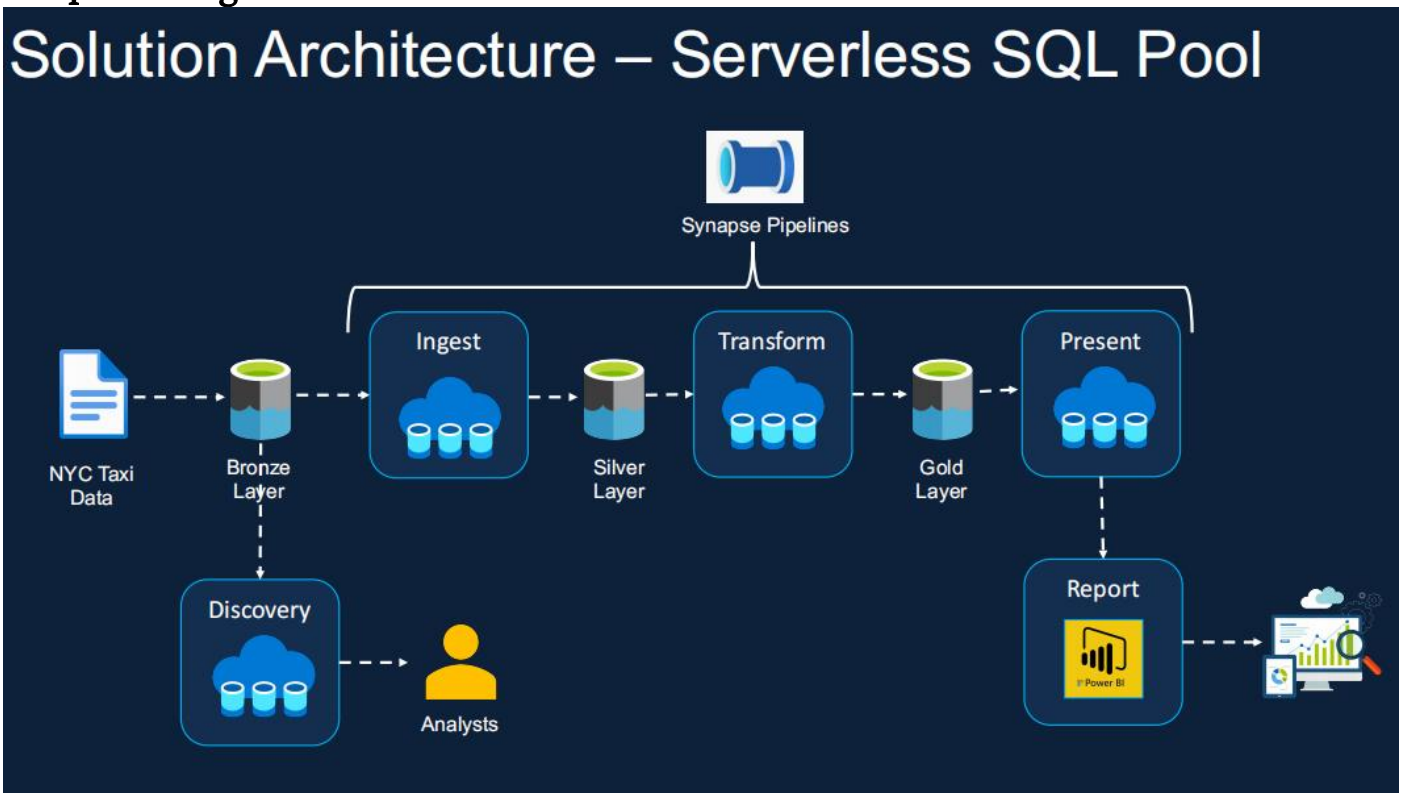
- Server-less SQL Pools
- Spark Pools
- Dedicated SQL Pools.

Step 2: Ingest NYC Taxi Data

Load raw data into Azure Data Lake Storage Gen2:

The data-set used in this project comes from the NYC Taxi and Limousine Commission (TLC), containing millions of records of taxi trips made in New York City. The data-set includes various features such as:

1. **Pickup and drop-off locations**
2. **Trip duration and distance**
3. **Fare amount, tips, and payment types**
4. **Passenger counts**



Step 3: Data Processing and Transformation with select statement

Exploring the Taxi Zone Data-set Using Server-less SQL Pool Compute Engine:

Query CSV Files

- We begin by creating an external data source in Azure Synapse Analytics. This configuration points the Server-less SQL Pool to the Azure Data Lake where our raw files are stored. The external data source acts as a reference to the remote storage, allowing us to query the data without having to import it into a database.

- The OPENROWSET function allows us to perform ad-hoc queries directly on the raw file stored in the data lake. By specifying the file path, format, and parser version, we can query the data without needing to load it into a formal table, enabling quick and cost-effective exploratory analysis.

- To ensure accuracy and avoid potential data mismatches, we define the expected schema for the data being queried. This includes specifying the appropriate data types for each column, which helps in properly interpreting the data and managing any potential collisions or conflicts in the data-set. By doing this, we ensure clean and structured results from the raw file.
- When dealing with raw data, it's common to encounter situations where a delimiter or special character (like a comma or quote) is embedded within the actual data values. To handle such cases, we use ESCAPECHAR and FIELDQUOTE in the query.
  1. ESCAPECHAR is used to specify a character that "escapes" special characters within the data, ensuring they are treated as part of the actual value and not as delimiters.
  2. FIELDQUOTE defines the character used to enclose fields that contain special characters, preventing misinterpretation of those values during data parsing.

- In cases where the fields in a file are separated by a delimiter other than the default (e.g., not a comma but a pipe | or semicolon ;), we use the FIELDTERMINATOR option. This allows us to define a custom character that acts as the delimiter between fields.

Query JSON Files

1. Reading Line-Delimited JSON Files

JSON document separated by a new line character (\n).

2. Reading Standard JSON Files

The standard JSON file, where multiple JSON documents are stored as a single array within one file. In this case, the JSON data is contained within square brackets ([]), with each document being part of the array.

3. Reading Classic Multi-Line JSON Files

Classic multi-line JSON files, also known as fully formatted JSON, span multiple lines and include a more structured representation of nested objects and arrays. Each JSON document might consist of several layers of data, making them more challenging to parse.

Extracting Elements Using JSON Functions

Using JSON_VALUE Function

The JSON_VALUE function allows us to extract simple scalar values from JSON documents and map them into SQL columns. However, this function has limitations—it cannot handle complex arrays and is inefficient when dealing with large JSON documents with multiple nested properties.

```
111
112  select CAST(JSON_VALUE(jsonread, '$.payment_type')as SMALLINT) payment_type,
113  CAST (JSON_VALUE(jsonread, '$.payment_type_desc[0].value') as VARCHAR(16)) payment_type_desc,
114  CAST (JSON_VALUE(jsonread, '$.payment_type_desc[1].value') as VARCHAR(16)) payment_type_describtion
115   from
116      OPENROWSET(
117          BULK 'raw/payment_type_array.json',
118          DATA_SOURCE = 'nyc_taxi_data',
119          FORMAT = 'csv',
120          FIELDTERMINATOR = '0x0b',
121          FIELDQUOTE = '0x0b'
122      )with(
123          jsonread NVARCHAR(MAX)
124      )
```

Results Messages

View Table Chart Export results

Search

payment_type	payment_type_desc	payment_type_describtion
1	Credit card	(NULL)
2	Cash	(NULL)

00:00:01 Query executed successfully.

Show hidden icons

Using OPENJSON Function

For more complex JSON documents, especially those containing arrays, the OPENJSON function is more suitable. This function can "explode" arrays, converting nested JSON objects into rows and columns. It is highly efficient for large JSON datasets, making it ideal for handling JSON files with multiple properties and arrays. This function provides more flexibility and scalability when working with complex JSON structures.

```
8       BULK 'raw/rate_code_multi_line.json',
9       DATA_SOURCE =  'nyc_taxi_data',
10      FORMAT = 'csv',
11      FIELDQUOTE = '0x0b',
12      FIELDTERMINATOR = '0x0b',
13      ROWTERMINATOR = '0x0b'
14  )with(
15      jsonmultiline NVARCHAR(MAX)
16  )
17  as result_json
18  CROSS APPLY openjson(jsonmultiline) with(
19      rate_code_id SMALLINT,
20      rate_code VARCHAR(50)
21  );
```

Results Messages

View Table Chart Export results

Search

rate_code_id	rate_code
1	Standard rate
2	JFK

00:00:00 Query executed successfully.

Querying Files with Wildcards: By using wildcards, it is possible to query multiple files within a folder without explicitly specifying each file name.

Recursive Wildcard Querying Across Sub-folders: Recursive wildcards provide the ability to query data from multiple sub-folders.

Utilizing Metadata Functions: Metadata functions, such as extracting the file name or file path, are used to retrieve detailed information about the source of the data being queried.

Aggregating Data by File: Aggregating data based on file-level metadata to gain insights into the structure and volume of the data in each file. Grouping data by file name or folder path helps in understanding data distribution and validating record counts across different files or time periods.

Extracting Folder Path Information: Metadata functions can also be used to extract specific segments of the folder path, such as year and month. Also used filter and grouped data based on folder structure, which is particularly useful when querying time-based or partitioned data stored in multiple directories.

Querying Parquet Files with Wildcards: This includes querying files for a specific year and month or applying recursive wildcards to extract data from all sub-directories, regardless of the file structure.

Defining Data Schema: When querying, the best practise is to specify the schema of the Parquet files by defining the columns and their data types. This ensures that the data is appropriately parsed and performance is optimized .

Filtering Based on Specific Time Periods in Delta Files: using where clause to retrieve data for specific time periods, such as a particular year and month, leveraging folder path metadata to refine query results of a Delta-format file select statement.

```
34  select top 100 *
35      from OPENROWSET(
36          BULK 'raw/trip_data_green_delta/',
37          DATA_SOURCE = 'nyc_taxi_data', FORMAT = 'Delta'
38      )with (
39      store_and_fwd_flag char(1) ,VendorID int,lpep_pickup_datetime datetime2(7),lpep_dropoff_datetime datetime2(7),
40      total_amount FLOAT,payment_type INT,trip_type INT,
41      year VARCHAR(5),
42      month VARCHAR(3) ) as result_set
43          where year = '2020' and month = '07';
```

Results Messages

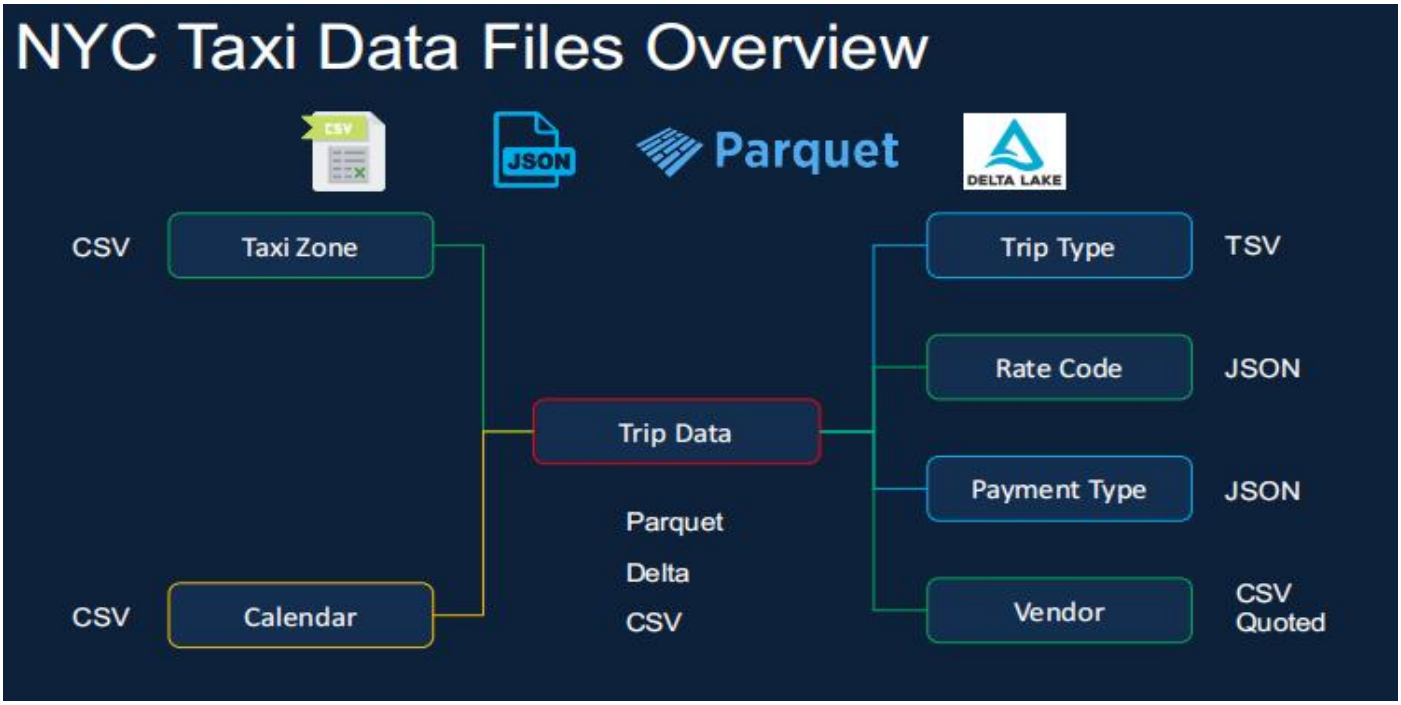
View Table Chart Export results

Search

store_and_fwd_flag	VendorID	lpep_pickup_datetime	lpep_dropoff_datetime	total_amount	payment_type	trip_type	year	month
N	2	2020-07-01T00:05:18.0000000	2020-07-01T00:22:07.0000000	21.8	2	1	2020	07
N	2	2020-07-01T00:47:06.0000000	2020-07-01T00:52:13.0000000	8.76	1	1	2020	07
N	2	2020-07-01T00:24:59.0000000	2020-07-01T00:35:18.0000000	10.3	2	1	2020	07
N	2	2020-07-01T00:55:12.0000000	2020-07-01T00:58:45.0000000	6.3	2	1	2020	07
N	2	2020-07-01T00:12:36.0000000	2020-07-01T00:20:14.0000000	12.05	2	1	2020	07
N	2	2020-07-01T00:30:55.0000000	2020-07-01T00:37:05.0000000	30.05	2	1	2020	07

00:00:08 Query executed successfully.

Ensuring data quality:



Performed a comprehensive data quality check on the trip dataset, focusing on key columns like **LocationID** and **total_amount**. Duplicates were identified and removed from the LocationID to ensure accuracy in the taxi zone data. The **total_amount** field was validated by calculating minimum, maximum, and average values to detect inconsistencies and outliers. Special attention was given to negative values, which were analyzed by **payment_type** to determine patterns and potential errors.

Additionally, trip duration were calculated by measuring the time difference between pickup and drop-off, categorizing trips into hourly intervals. This provided insights into trip length distribution, essential for operational analysis and improving service efficiency.

Combining Data from Different Sources:

For geographic insights, trip data from **Parquet files** was combined with NYC taxi zone data from a **CSV file**. This enabled analysis of trip counts by borough, revealing ride distribution patterns and ranking boroughs by trip frequency.

Lastly, a final query summarized trips by duration, enabling a deeper understanding of taxi demand trends and customer behavior based on trip length. This analysis helps identify high-demand periods for both short and long trips.

Step 4: Data Analysis

```
select
zones_data.Borough, count(*) as total_trip,
COUNT(CASE when paymenttype_data.payment_type_desc = 'Credit card' then trips_data.VendorID end) as card_trips,
COUNT(CASE when paymenttype_data.payment_type_desc = 'cash' then trips_data.VendorID end) as cash_trips,
cast((sum(CASE when paymenttype_data.payment_type_desc = 'cash' then 1 else 0 end)/
cast(count(trips_data.VendorID)as decimal))
*100 as decimal(5,2))as cash_trips_percentage,
cast((sum(CASE when paymenttype_data.payment_type_desc = 'Credit card' then 1 else 0 end)/
cast(count(trips_data.VendorID)as decimal))
*100 as decimal(5,2))as card_trips_percentage

```

Results Messages

View Table Chart Export results

Search

Borough	total_trip	card_trips	cash_trips	cash_trips_percentage	card_trips_percentage
EWR	33	29	3	9.38	90.63
Staten Island	2817	1770	243	11.99	87.32
Unknown	6577	3316	1918	35.89	62.05
Bronx	262014	41628	34282	44.71	54.29
Queens	526912	174594	182593	50.68	48.46

00:00:05 Query executed successfully.

This segment of the project, focused on analyzing payment methods used in NYC Green Taxi trips. This analysis highlights the relationship between trip locations, payment types, and overall trip counts. Here's a detailed overview of the steps taken:

Aggregating Trip Data by Borough:

executed a query to count the total number of trips in each borough. By aggregating the trip data based on the borough, I was able to capture the overall trip volume, which provides insights into which areas of New York City are most active in terms of taxi service usage.

Classifying Trips by Payment Type:

The query specifically classified the trips based on payment methods, categorizing them into credit card and cash transactions. This classification was achieved using conditional counting, allowing me to isolate and count trips that were paid for with each method. The result is a clearer picture of customer preferences regarding payment types in different boroughs.

Calculating Payment Type Percentages:

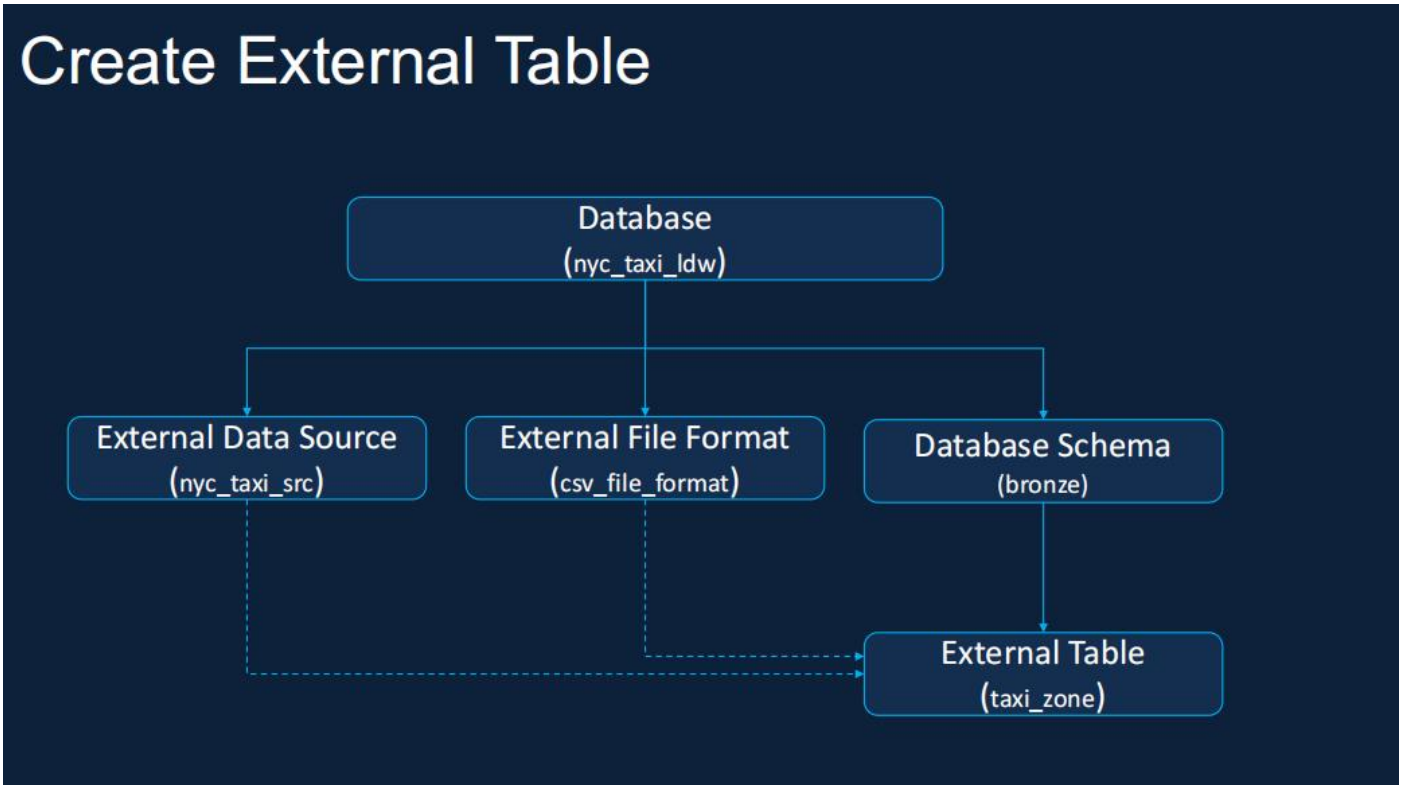
To gain further insights, calculated the percentage of trips paid for with cash and credit cards relative to the total number of trips. This was done by summing the relevant counts and dividing by the total trip count, providing a percentage value formatted to two decimal places. This percentage helps to understand how much each payment type contributes to the overall taxi service in each borough.

Joining Multiple Data Sources:

The query involved joining data from three different sources:
Payment Type Data: Loaded from a JSON file, this data provided details on payment methods and descriptions.
Trip Data: This data was sourced from a Delta format file containing trip details, including pickup location and payment type.
Taxi Zone Data: A CSV file provided information on the borough corresponding to each pickup location.

Step 5: Data Visualization

we implemented a logical data warehouse (DWH) for NYC Taxi data.
The data warehouse follows the bronze, silver, and gold layers architecture, which aligns with the data lake storage strategy. Below are the key actions taken in the project:



The solution was structured into three layers:

- **Bronze: Raw data ingestion.**

1. Database and Schema Setup

created a new database named logical_DWH_NYC_taxi and set its collation to Latin1_General_100_BIN2_UTF8 to ensure consistent string encoding.

2. External Data Source and File Formats

An external data source named logical_DWH_exDS was created, pointing to an Azure Data Lake Storage Gen2 location containing the NYC Taxi data.

3. **Several external file formats** were defined to handle different data types:

- CSV: For structured tabular data (e.g., taxi trip data).
- TSV: For tab-delimited data.
- Parquet: For compressed columnar data.
- Delta: For Delta Lake format .

4. External Tables in the Bronze Layer

External tables were created under the Bronze schema to represent raw data ingested directly from the external source.
The reject options were configured to handle erroneous rows during ingestion, with rejected rows stored in specific locations for later analysis.

5. Views on External Data

created views to extract structured data from JSON files in the external storage
These views utilize the OPENROWSET function to read and parse JSON data stored in the external data source.

6. Partitioned Data Handling

A view named bronze.view_partition_tripdata was created to handle partitioned taxi trip data, which is stored by year and month.

The OPENROWSET function was used to dynamically reference the data partitions, allowing for efficient querying of large datasets.
The view includes additional columns (year, month) extracted from the file path to make filtering and analysis easier.
We performed sample queries to retrieve data for specific partitions (e.g., trip data from July 2021).

- # Silver: Cleaned and transformed data.

Data was selected from the respective tables in the bronze layer and written into the parquet format in the silver layer.
JSON data for rate_code was read and parsed from raw JSON files using OPENROWSET and transformed before being written to parquet while JSON data for payment_type was selected from the respective view in the bronze layer and written into the parquet format in the silver layer.

External Table Definitions:

Each external table in the silver layer was defined with Location Points to the respective folder in the silver layer storage where the transformed parquet files are stored.

Stored procedures:

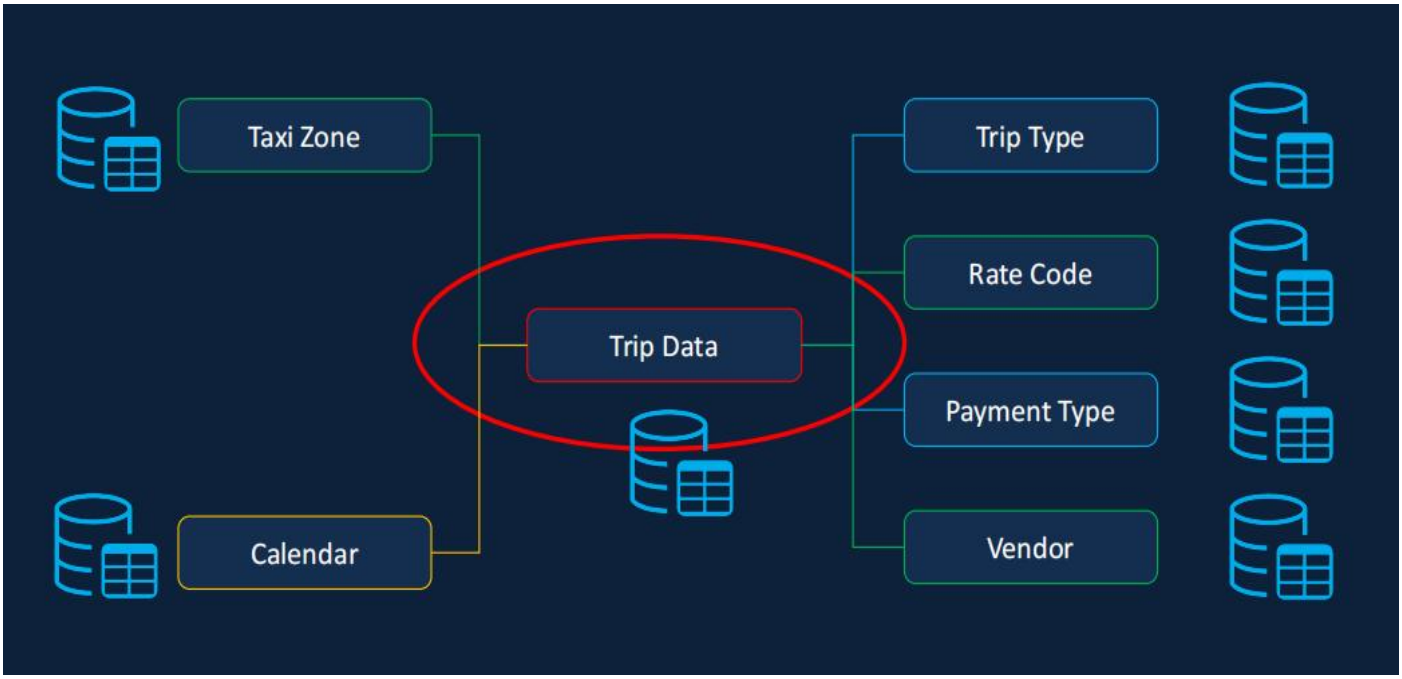
One key challenge with handling multiple files from folders and subfolders is that it lead to the writing of all data into a single parquet file without logical partitioning. This can significantly hinder performance when querying large datasets, as the entire file must be scanned to retrieve specific data.

To address this, a stored procedure was developed that dynamically partitions the trip data based on the input parameters for year and month. This enables the segregation of data into smaller, logically organized parquet files, significantly improving query performance and scalability.

Key Functionality:

- Dynamic Partitioning: The procedure partitions the data for each specified year and month, organizing it into structured directories (year=YYYY/month=MM) in the silver layer.
- Efficient Table Management: If a partition table for a specific year and month already exists, the procedure automatically drops it before creating a new one, ensuring a clean slate for data ingestion.
- Bronze to Silver Data Flow: Data is extracted from the bronze layer and written into corresponding parquet files in the silver layer, allowing for optimized data storage and retrieval.

This partitioning approach ensures that each year and month is stored separately, improving scalability and enabling faster query execution. Queries that filter data by time periods (e.g., a specific year or month) can now efficiently scan only the relevant partitions, rather than processing the entire data-set.



- ## Gold: Aggregated and refined data for analytics.

The gold layer is designed specifically for reporting and analysis. It contains aggregated and processed data that delivers valuable insights for decision-making. This layer is optimized to support efficient queries for business intelligence purposes.

1. Stored Procedure for Final Reporting Table:

A stored procedure was developed to dynamically build final reporting tables based on the specified year and month. For each year and month combination, the procedure performs the following steps:

1. Creation of External Table:

The procedure creates an external table that stores aggregated trip data in the gold layer, organized by year and month.
Data is written in Parquet format to ensure efficient storage and fast access.

2. Data Aggregation:

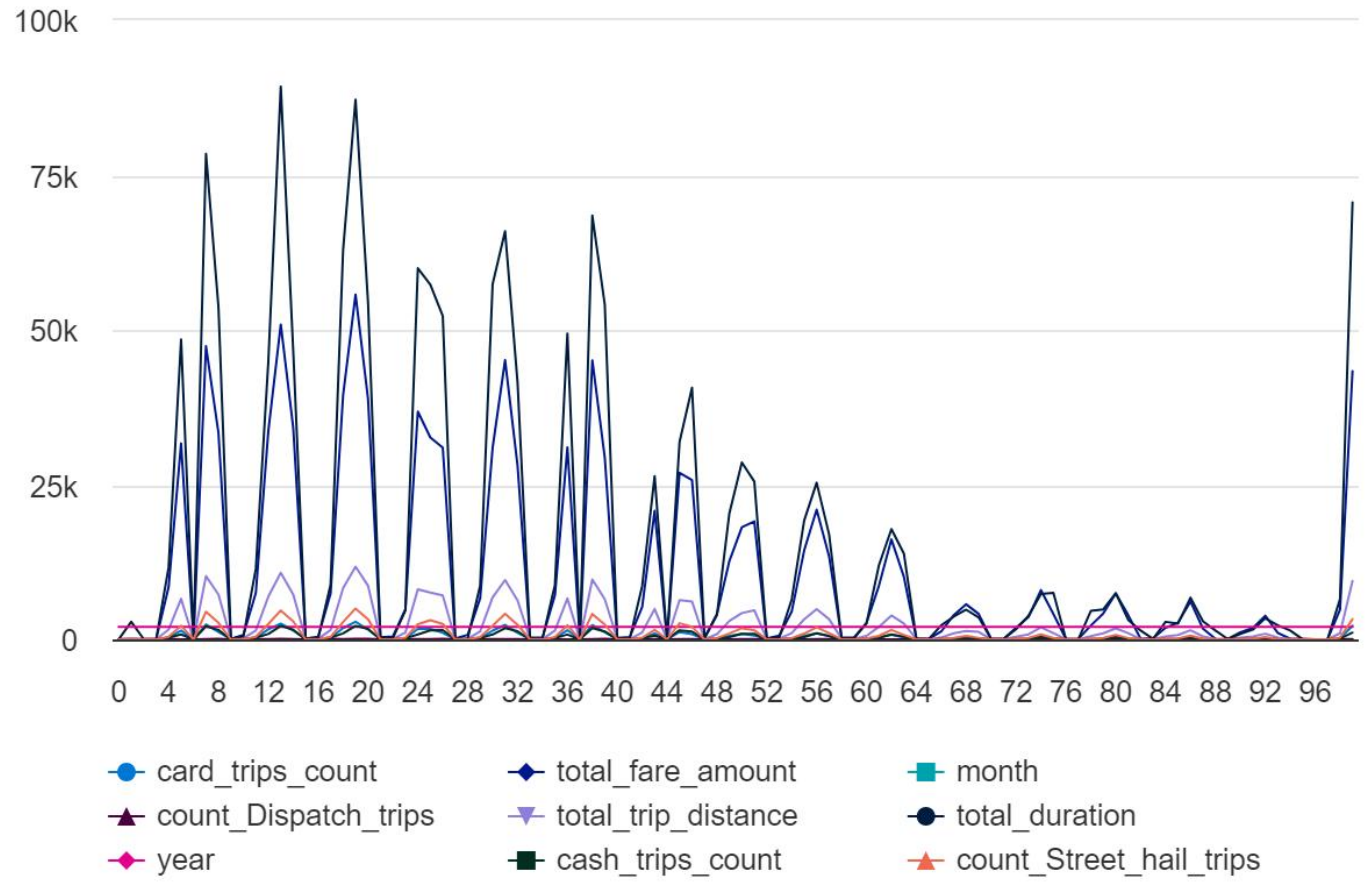
- The procedure aggregates key metrics from the silver layer, including:
- Total trips by payment type (e.g., credit card, cash).
- Total trips by trip type (e.g., Dispatch, Street-hail).
- Total trip distance and total fare amount.
- Total trip duration in minutes.
- The data is joined with dimensional tables, such as taxi zones, calendar, payment types, and trip types, to enrich the reporting data.
- This process ensures the trip data is efficiently organized for querying by partitioning it based on year and month.

2. Gold Layer View for Reporting:

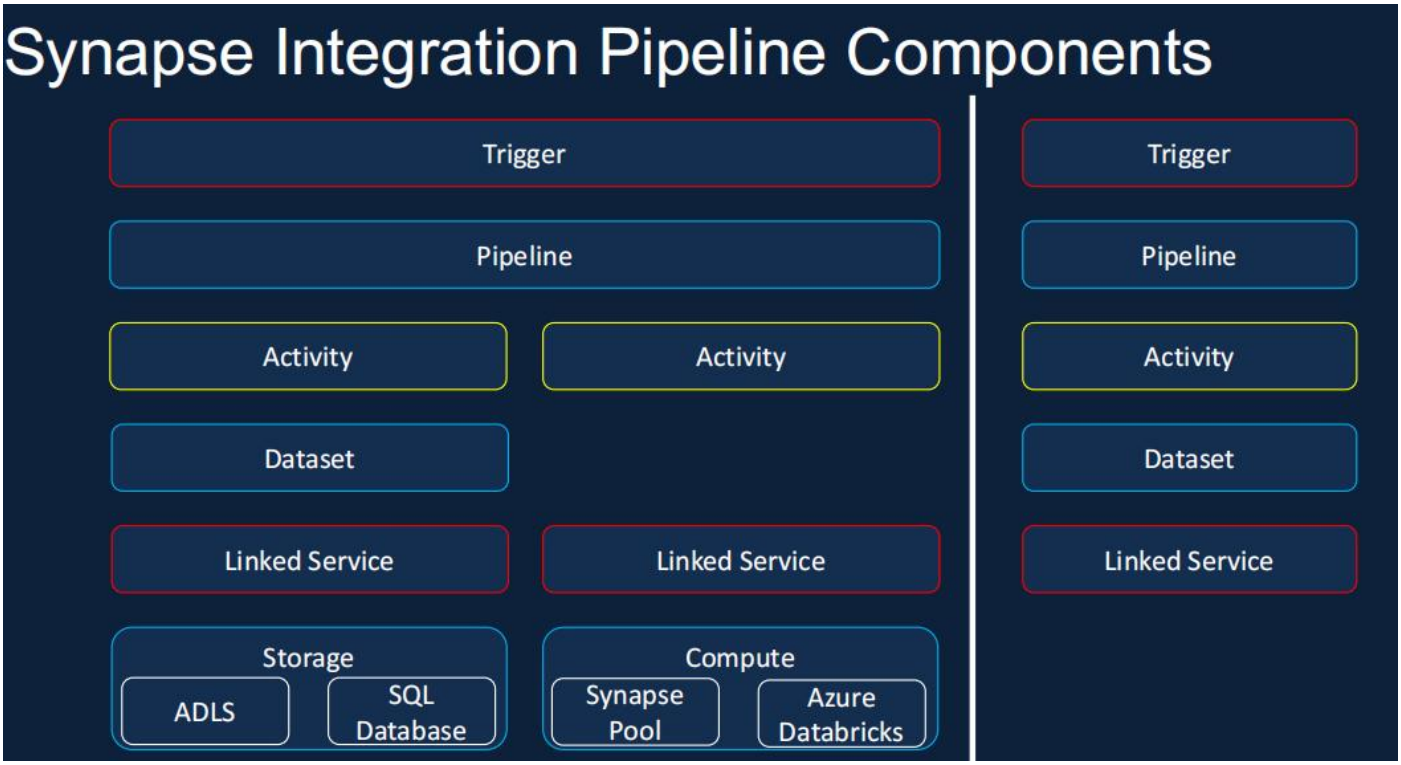
A unified view was created to facilitate easy querying of the aggregated data stored in the gold layer. This view provides a consolidated interface that presents key reporting metrics, such as:

- Trip Date: The date of the trip.
- Borough: The location of the trip.
- Day of the Week: Indicates whether the trip occurred on a weekday or weekend.
- Trip Counts: Number of trips categorized by payment type (credit card, cash) and trip type (Dispatch, Street-hail).
- Trip Distance: Total distance covered by trips.
- Fare Amount: Total fare generated.
- Trip Duration: Total duration of trips in minutes.

This view enables users to easily retrieve the metrics required for analysis and visualization, ensuring efficient and scalable reporting across the data-set..



Step 6: Synapse Pipelines



To automate the creation of the silver layer table for the taxi_zones data, a Synapse pipeline was developed. This pipeline ensures that the existing data is deleted and the table is recreated with updated data from the bronze layer.

Delete Activity:

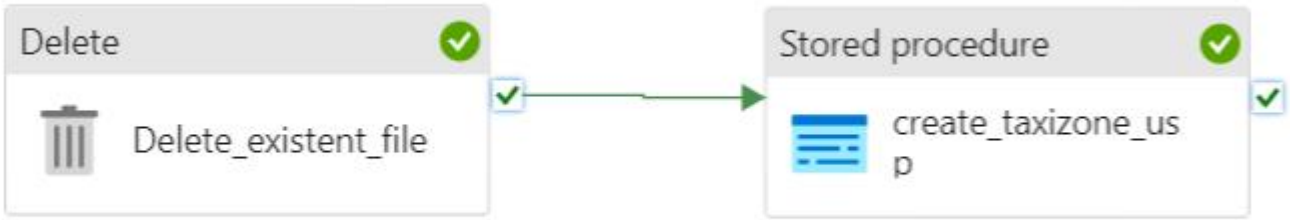
responsible for deleting the folder at the specified location as with just the SQL code, we can't directly delete files or folders from the storage layer.
The Delete Activity ensures that old data in the silver layer is completely removed before new data is written.

Script Activity:

It performs the following operations:
Drop Table if Exists: The existing table in the silver schema is dropped if it already exists. This ensures that stale metadata is removed.
Create External Table: A new external table is created in the silver layer, sourcing data from the bronze.taxi_zones table.

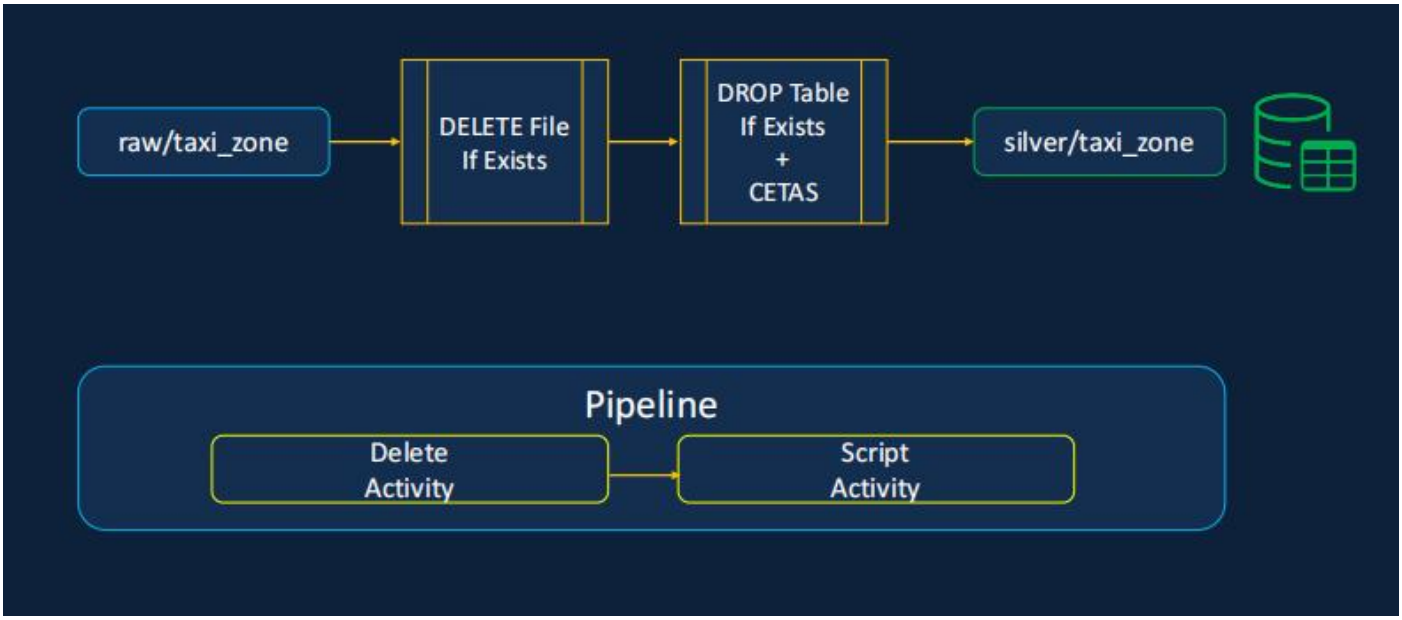
Stored Procedure Activity:

The pipeline uses the Stored Procedure Activity to call the stored procedure with the same script.
This ensures that the table is dynamically dropped and recreated using the latest data from the bronze layer.



Using Variables and Parameters for Re-usability:

By incorporating parameters and variables, this setup allows you to reuse the same pipeline across different datasets or folders without the need to hard-code values. You simply provide new parameter values or change the variable content, and the pipeline will work for different scenarios.



Dynamic Pipeline:

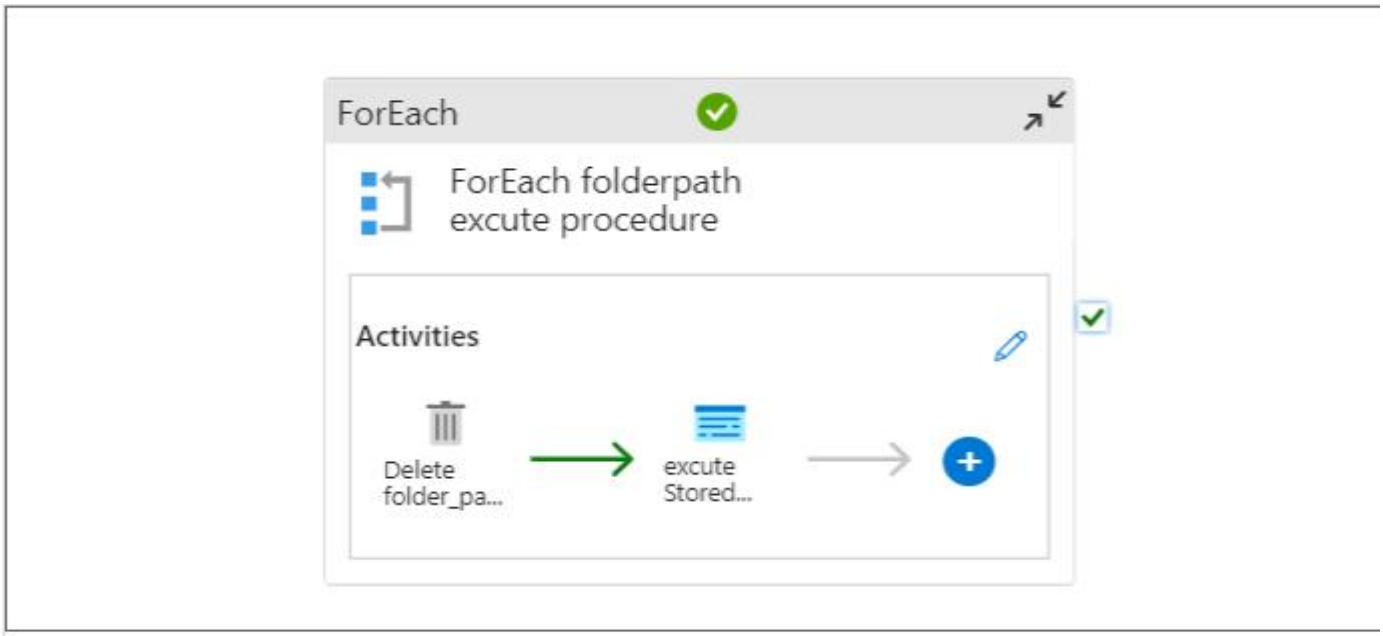
1. variable arrays:

Automated the pipeline using a ForEach activity that iterates through an array of folder paths and procedure names, you followed a great approach by using variable arrays.

ForEach Activity: Loops over each object in the array, processing one folder and stored procedure pair at a time.

Delete Activity: Deletes the folder specified by the current item in the array.

Stored Procedure Activity: Executes the stored procedure associated with the current folder to recreate or load data into the table.

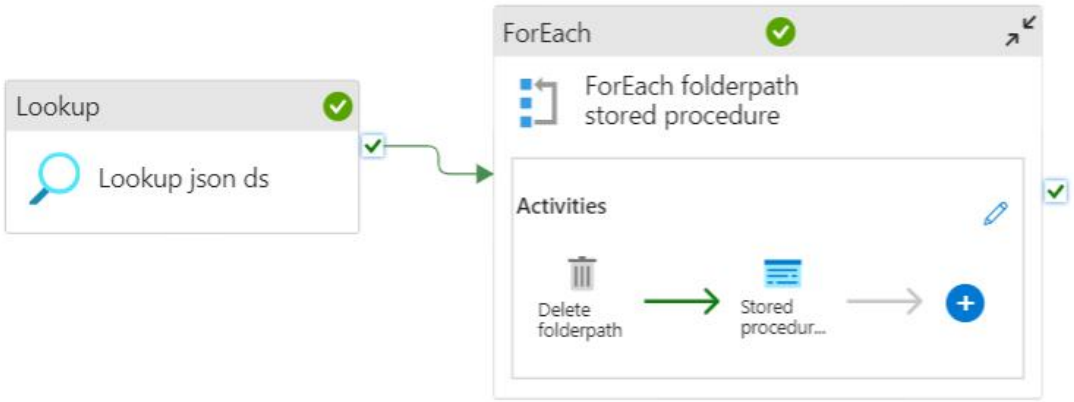


Parameters Variables Settings Output

excute Stored procedure	Succeeded	Stored procedure	9/30/
Delete folder_path	Succeeded	Delete	9/30/
excute Stored procedure	Succeeded	Stored procedure	9/30/
Delete folder_path	Succeeded	Delete	9/30/
excute Stored procedure	Succeeded	Stored procedure	9/30/
Delete folder_path	Succeeded	Delete	9/30/
ForEach folderpath excute pr...	Succeeded	ForEach	9/30/

2. Lookup Activity:

stored the folder paths and procedure names in a JSON file and added them to a data-set so that we can retrieve them dynamically using a Lookup Activity.
Lookup Activity: Retrieves the list of folder paths and procedure names from the data source.
ForEach Activity: Iterates over the retrieved values.
Delete Activity: Deletes the folder specified by the current item in the loop.
Stored Procedure Activity: Executes the corresponding stored procedure for the folder.



Parameters Variables Settings Output

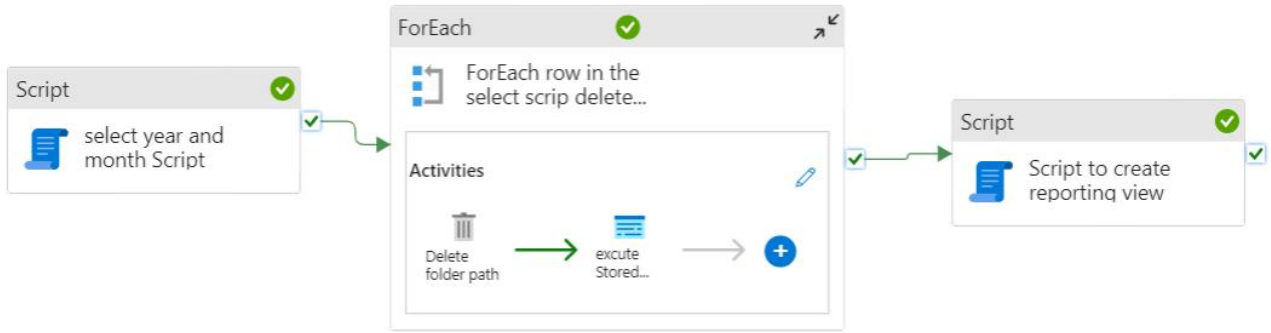
Delete folderpath	Succeeded	Delete	9/30/2024, 4:12:18 PM	4s
Stored procedure exution	Succeeded	Stored procedure	9/30/2024, 4:12:05 PM	11s
Delete folderpath	Succeeded	Delete	9/30/2024, 4:11:40 PM	24s
Stored procedure exution	Succeeded	Stored procedure	9/30/2024, 4:11:31 PM	9s
Delete folderpath	Succeeded	Delete	9/30/2024, 4:11:24 PM	6s
ForEach folderpath stored pr...	Succeeded	ForEach	9/30/2024, 4:11:24 PM	1m 56s
Lookup json ds	Succeeded	Lookup	9/30/2024, 4:11:07 PM	17s

3. Partitioned File Pipeline:

Script Activity runs the SQL query and returns the distinct year and month from the trip data.
The ForEach activity iterates over the year and month combinations returned by the Script Activity.

For each iteration:

- The Delete Activity deletes the folder corresponding to the current year and month.
- The Stored Procedure Activity runs the stored procedure for the current year and month to populate the gold layer.
- an additional Script Activity to manage the creation of the final reporting view in the gold layer by querying aggregated data



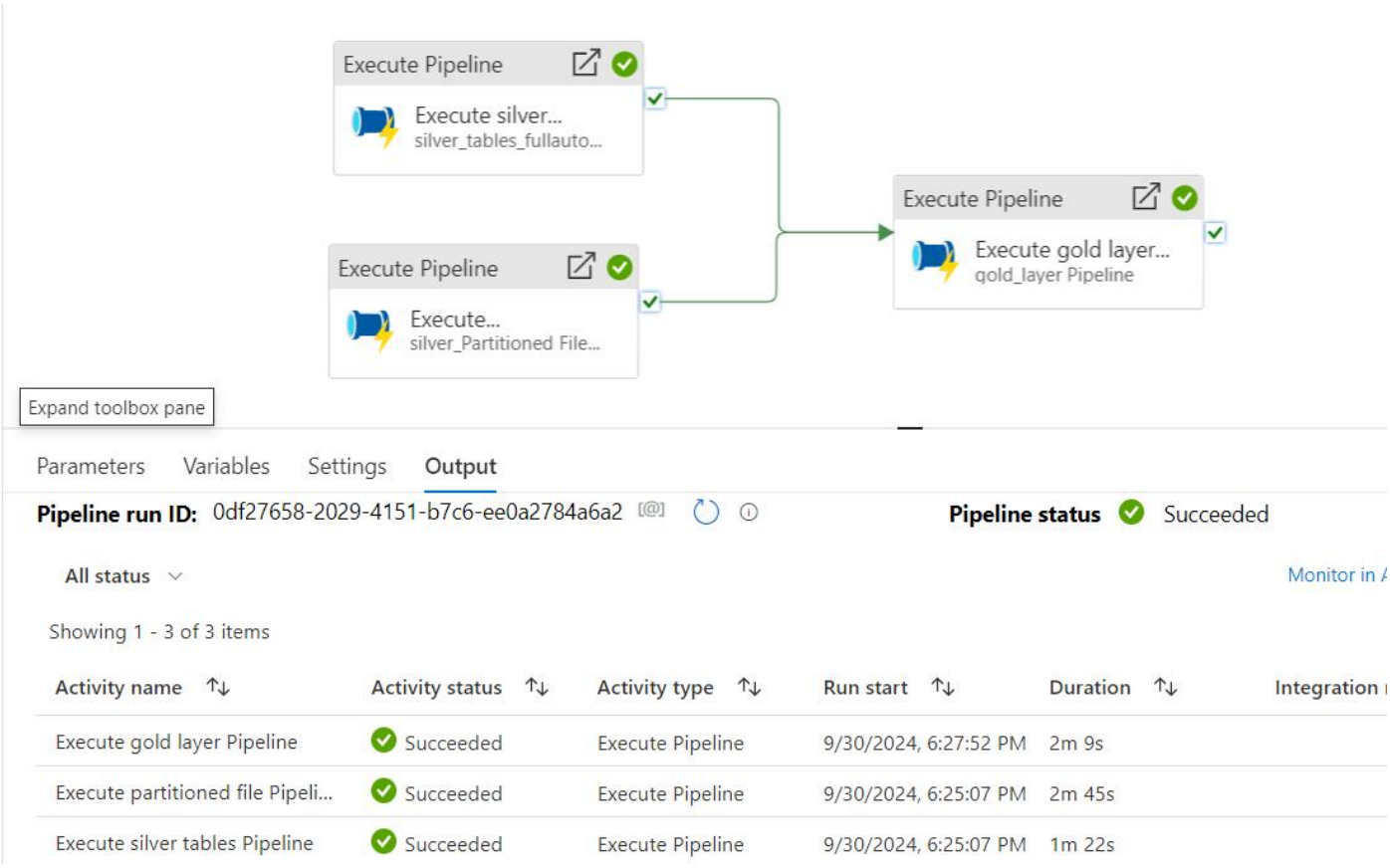
Parameters Variables Settings Output

Script to create reporting view	Succeeded	Script	9/30/2024, 6:14:18 PM	13s	AutoResolveIntegrati
excute Stored procedure	Succeeded	Stored procedure	9/30/2024, 6:14:08 PM	7s	AutoResolveIntegrati
excute Stored procedure	Succeeded	Stored procedure	9/30/2024, 6:13:58 PM	9s	AutoResolveIntegrati
excute Stored procedure	Succeeded	Stored procedure	9/30/2024, 6:13:58 PM	8s	AutoResolveIntegrati
Delete folder path	Succeeded	Delete	9/30/2024, 6:13:53 PM	14s	AutoResolveIntegrati
Delete folder path	Succeeded	Delete	9/30/2024, 6:13:52 PM	5s	AutoResolveIntegrati
excute Stored procedure	Succeeded	Stored procedure	9/30/2024, 6:13:49 PM	7s	AutoResolveIntegrati

4. Pipeline Dependencies Activity:

To ensure a smooth and efficient data processing workflow, incorporated Pipeline Dependencies, creating a well-structured execution sequence.

- Execute the Silver Tables Pipeline.
- Execute the Partitioned Tables Pipeline.
- After both the above pipelines finish, the Gold Layer Pipeline is triggered automatically.
- Implemented a schedule trigger to automate your pipeline execution every 24 hours, which ensures your data processing workflow stays up to date without manual intervention.



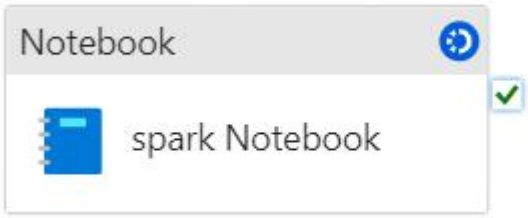
Step 7: Spark Pool

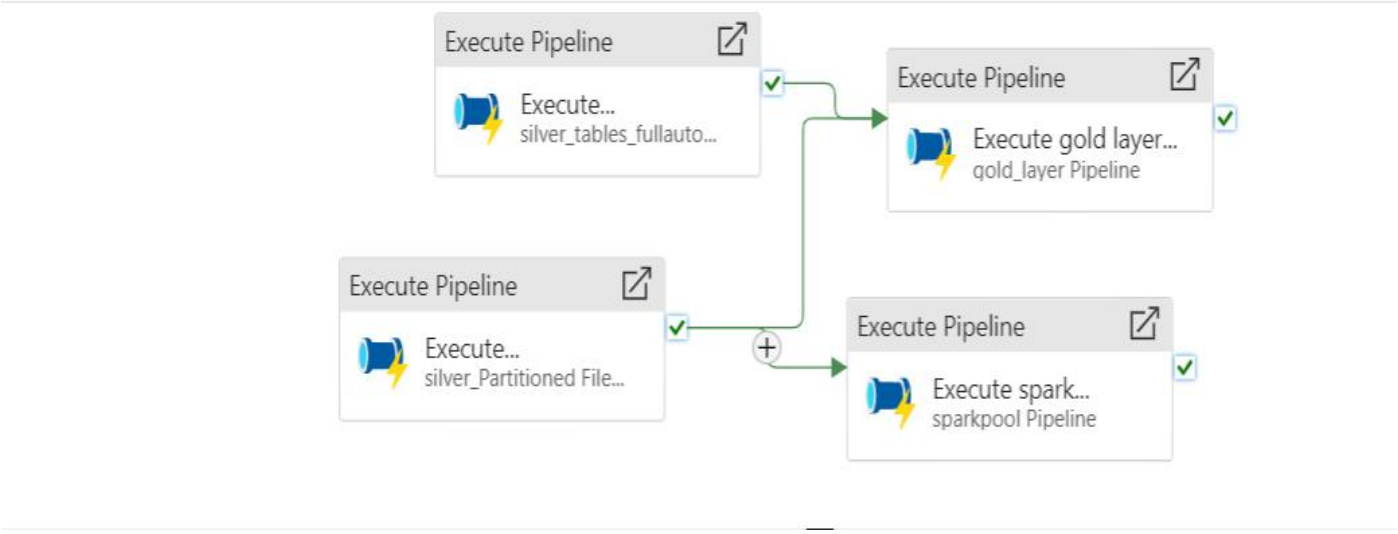
Exploring the Taxi Zone Data-set Using Spark Pool:

- Created a new notebook using Spark code to Performs aggregations to calculate total trip counts and total fare amounts by grouping the data based on specified columns and then Writes the aggregated Data-frame back to a gold table, partitioning the data by year and month.

- Before integrating it into the pipeline, ran the notebook independently to ensure that it functions as expected and produces the desired output.

- Used the Notebook Activity to add Spark notebook to a pipeline.
- Added another execute pipeline activity to the full process pipeline so it can execute after Executing the Partitioned Tables Pipeline.

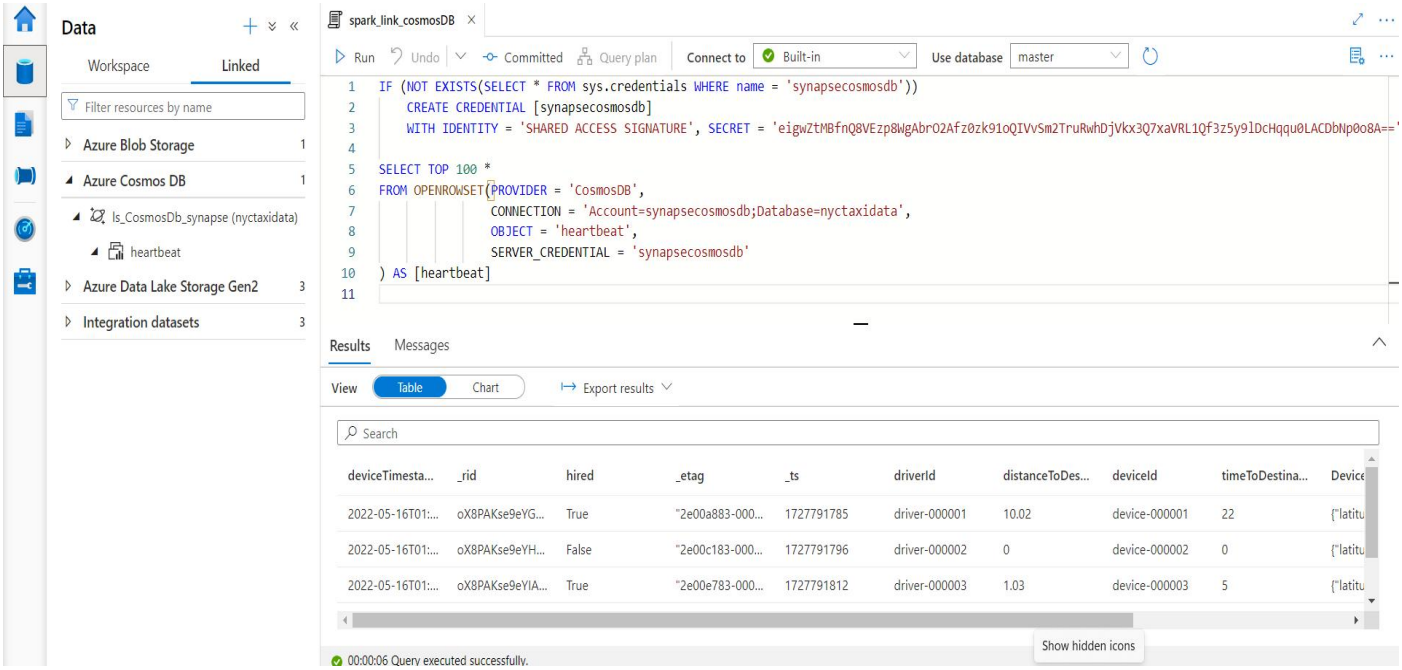


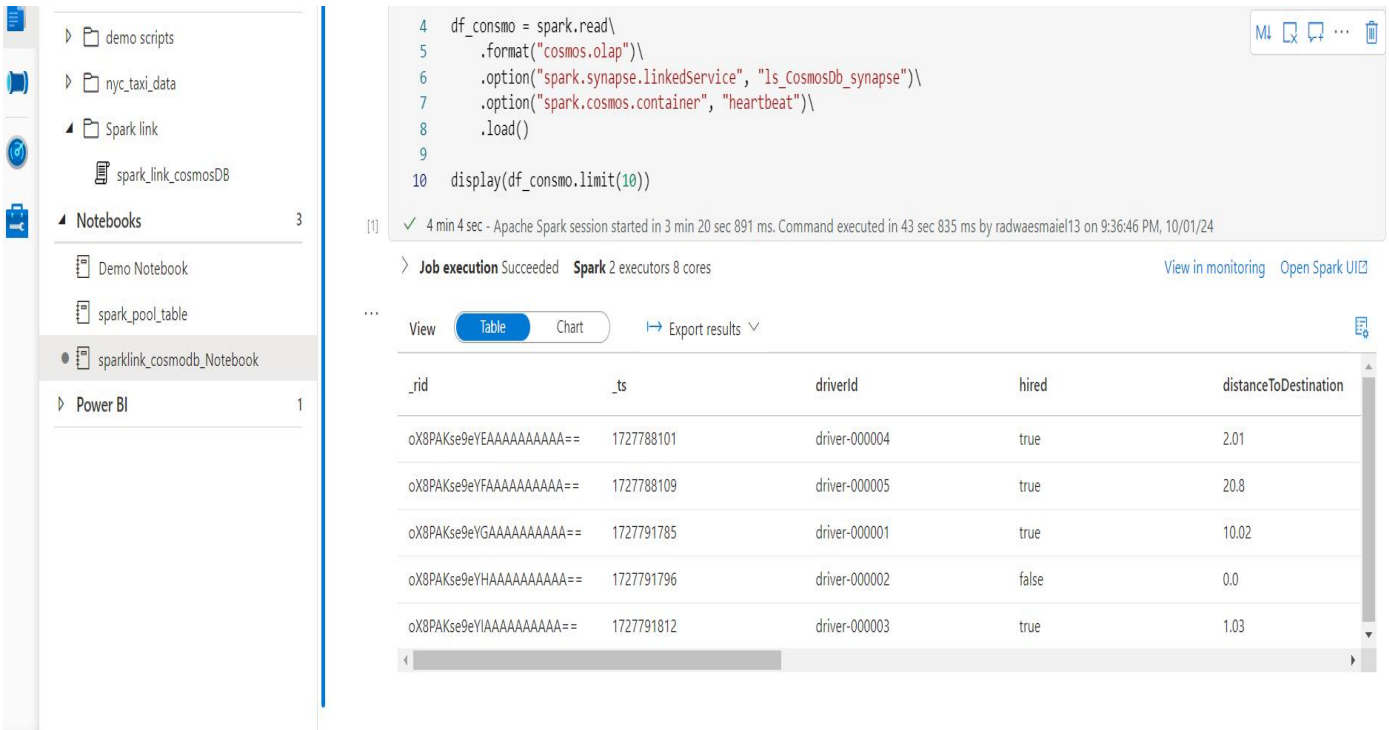


Step 8: Synapse Link

- created a Cosmos DB account and filled a container with sample driver-related data, including information about drivers, their devices, and their destinations.
- connected Cosmos DB to Azure Synapse Analytics, enabling to query Cosmos DB data via the openrowset function in Synapse SQL on a server-less pool.
- used Azure Synapse's Spark pools to read data from Cosmos DB's analytical store directly into a Spark DataFrame.

-





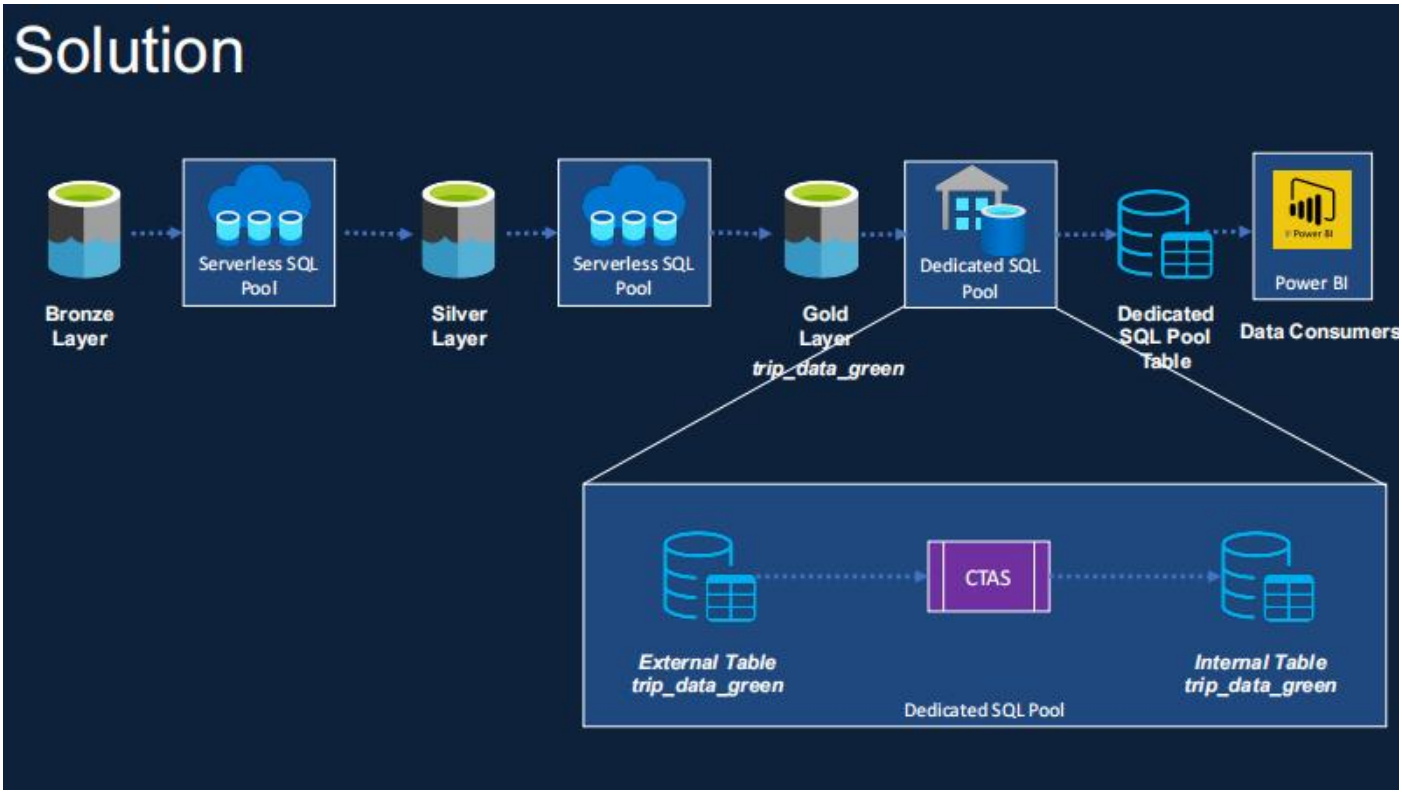
Step 9: Dedicated SQL Pool

Exploring the Taxi Zone Data-set Using Dedicated SQL Pool:

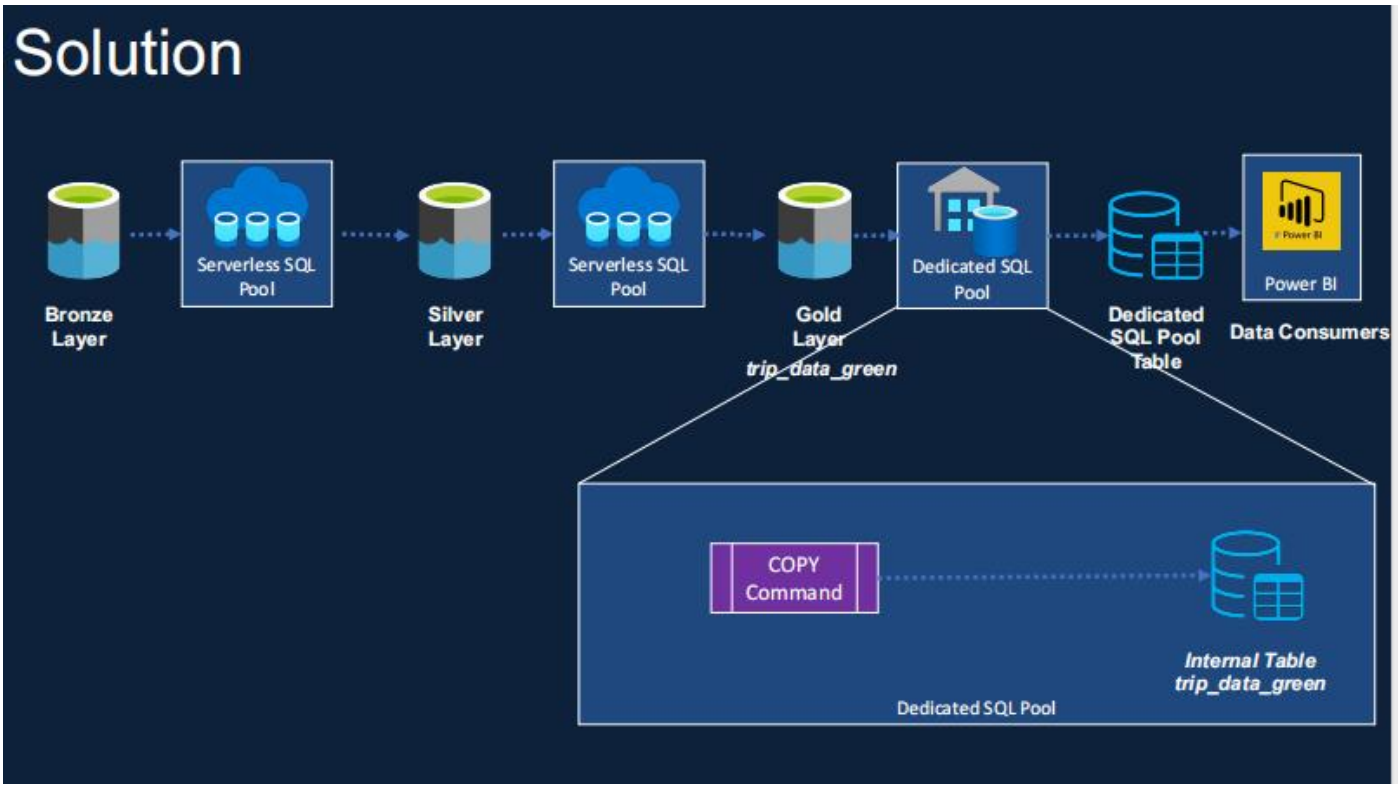
Sat up a full pipeline for loading data from Azure Data Lake Storage (ADLS) into Azure Synapse Analytics Dedicated SQL Pool by following these key steps:

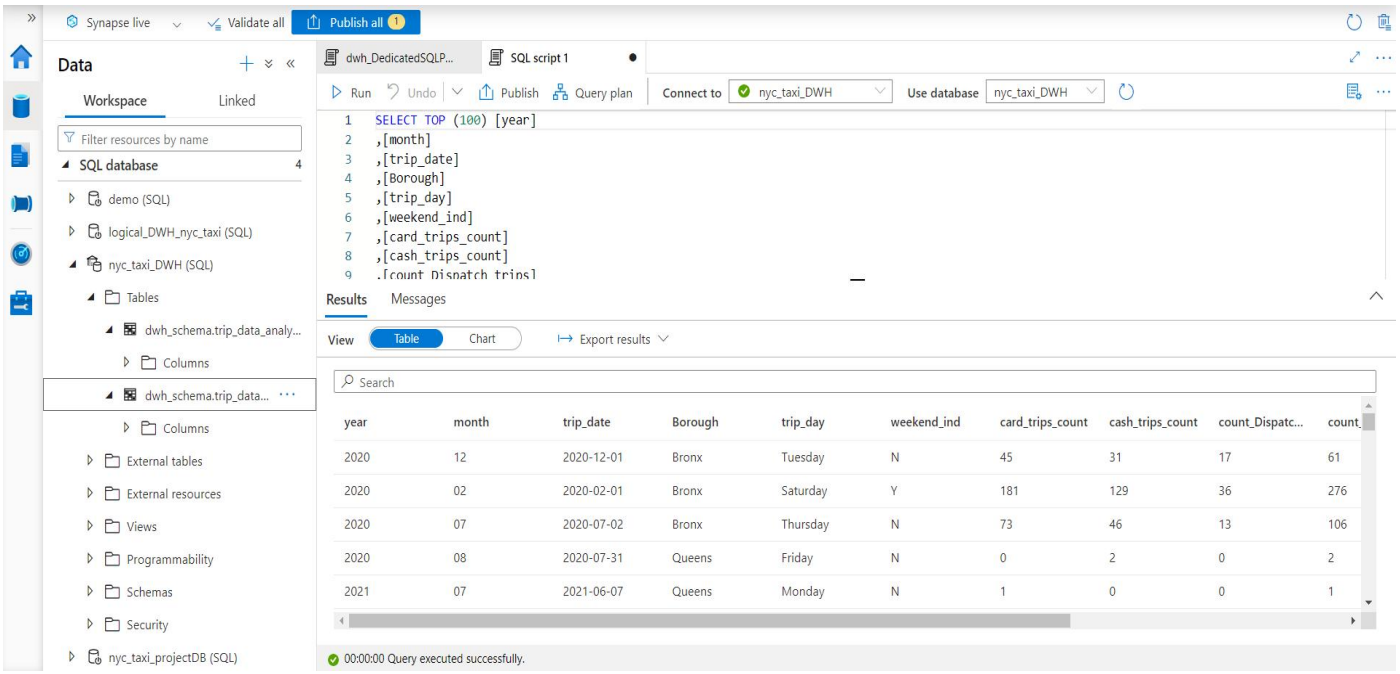
1. Creating External and Internal Tables:

Created external tables in the staging schema, which read directly from Parquet files stored in the gold layer of your data lake. used the CREATE TABLE AS SELECT (CTAS) statement to copy the data from the external tables into the dedicated SQL pool tables for faster querying and analytics purposes. This approach effectively makes dedicated SQL pool the serving layer.



An alternative method was implemented is the COPY command in Synapse, which directly loads data from ADLS into the dedicated SQL pool, bypassing the need for external tables. The COPY command simplifies the process and supports file formats like Parquet it efficiently moved data into internal tables, improving performance for frequently accessed data.



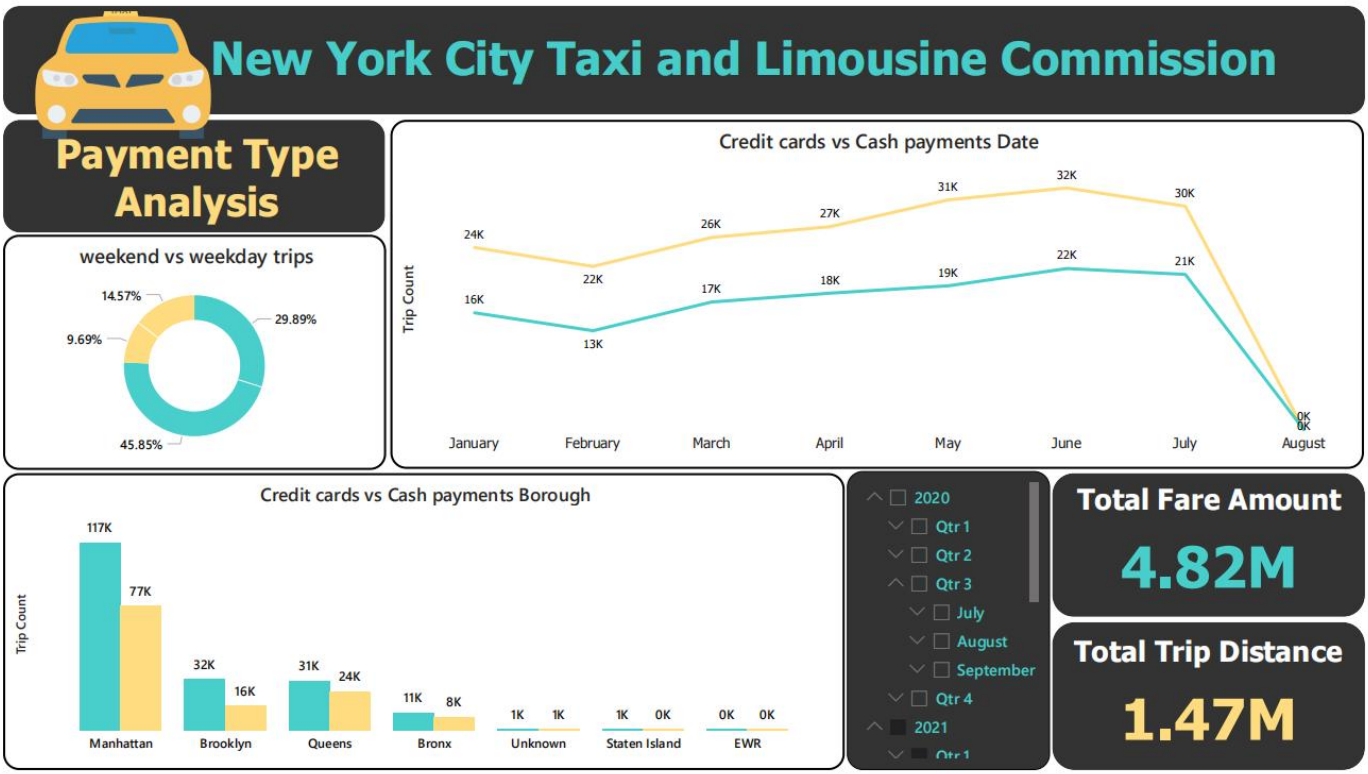


Step 10: Power BI Integration

First report requirements :

The NYC Taxi and Limousine Commission aims to increase credit card usage to 90% of all payments, reducing cash payments to less than 10%. To support this, they require:

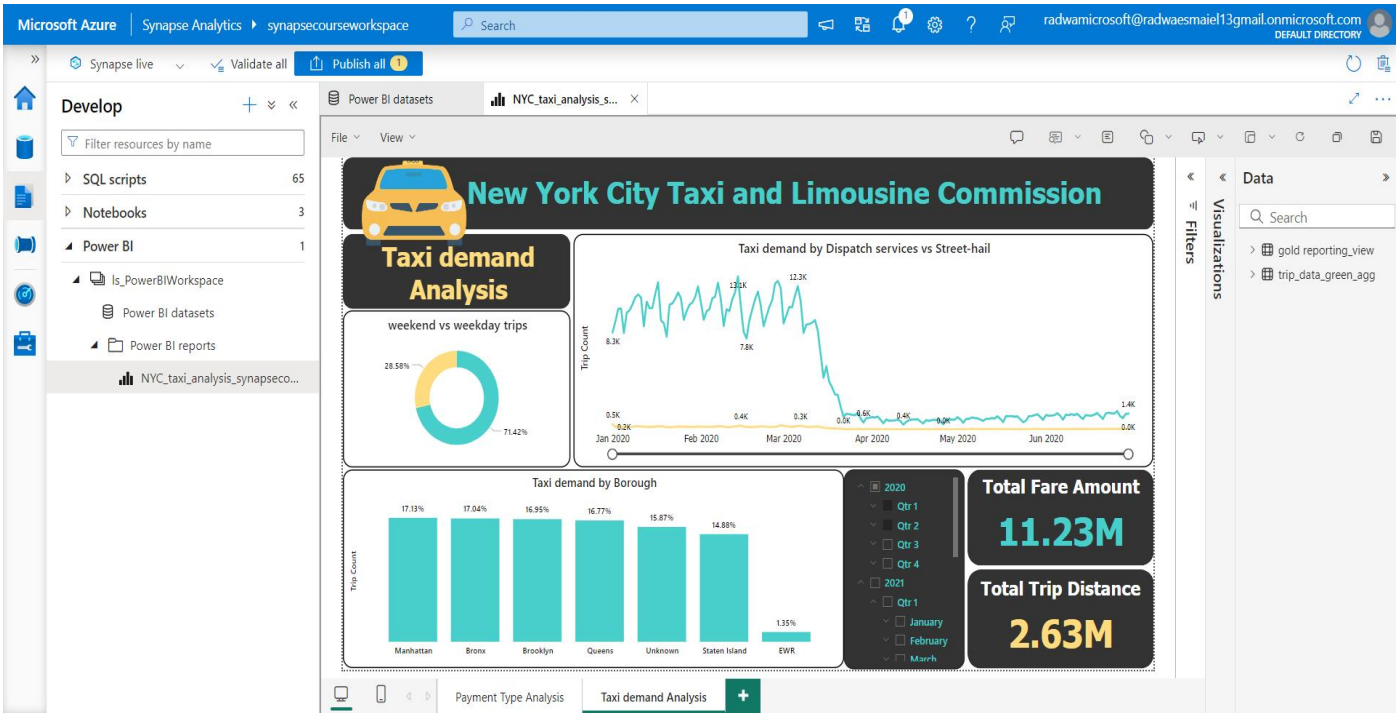
- Current Payment Breakdown: Provide data on the ratio of trips paid by cash vs. credit card.
- Day-Specific Payment Behavior: Analyze payment patterns by day, identifying differences between weekdays and weekends to optimize campaign targeting.
- Borough-Level Insights: Break down payment behavior by borough to identify areas with low card usage for more targeted campaigns.



Second report requirements :

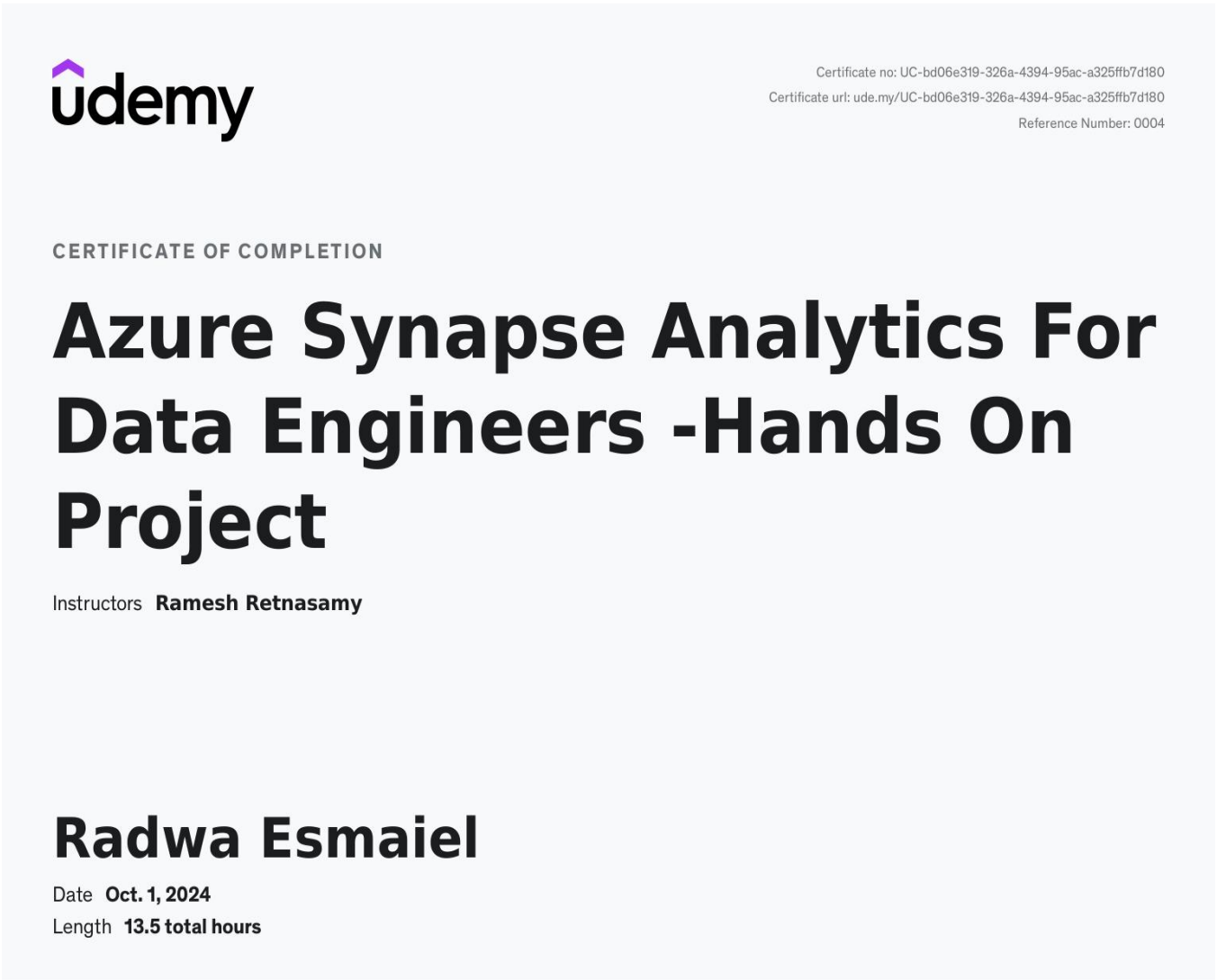
The NYC Taxi and Limousine Commission seeks to identify taxi demand to adjust license supply accordingly. They require:

- Borough-Based Demand: Analyze demand trends across different boroughs.
- Day-Specific Demand: Track taxi demand by day of the week.
- Dispatch Service Insight: Determine if customers using dispatch services are getting taxis retail.
- Trip Metrics: Provide total trip distance, and fare, broken down by trip, date, and borough.



course and use case: https://www.udemy.com/course/azure-synapse-analytics-for-data-engineers/

Course certificate


udemy

Certificate no: UC-bd06e319-326a-4394-95ac-a325ffb7d180
Certificate url: ude.my/UC-bd06e319-326a-4394-95ac-a325ffb7d180
Reference Number: 0004

CERTIFICATE OF COMPLETION

Azure Synapse Analytics For Data Engineers -Hands On Project

Instructors **Ramesh Retnasamy**

Radwa Esmaiel

Date **Oct. 1, 2024**
Length **13.5 total hours**

GitHub: link of full project and Data :

https://github.com/RadwaEsamiel/Azure-Synapse-Analytics.git