Analyse Yelp Dataset with Spark & Parquet Format on Azure Databricks

CookBook

ProjectPro

Table of Content:

Introduction

In this project, we shall execute a data pipeline to process and analyze Yelp's raw community-driven JSON dataset using Microsoft Azure services, aiming to derive actionable business insights and improve customer engagement. The pipeline is designed within a cloud-based, scalable architecture, ensuring seamless orchestration, efficient data transformation, and high-performance analytics.

This project introduces a robust, end-to-end Azure-based data pipeline enabling standardized data processing, sentiment analysis, and performance benchmarking for businesses listed on Yelp. By leveraging Azure services such as Data Lake Storage Gen2, Data Factory, and Databricks, we will build a powerful framework capable of handling large-scale JSON data, performing complex transformations, and delivering real-time business intelligence.

Use Cases for the Proposed Pipeline

1. **Business Performance and Trend Analysis**

   Organizations can analyze patterns across various regions and categories by transforming raw business data into structured formats. This pipeline extracts JSON data from Yelp, stores it in Azure Data Lake Storage Gen2, and uses Azure Databricks to identify top-performing businesses, emerging categories, and market trends. The processed output enables detailed benchmarking and strategic decision-making.

2. **Customer Sentiment and Review Mining**

   By processing review data using Spark-based analytics in Azure Databricks, this pipeline uncovers sentiment trends over time, tracks customer satisfaction, and identifies service gaps. This empowers businesses to adapt based on real feedback and market perception proactively.

3. **User Behavior Insights and Check-In Patterns**

   The system aggregates user activity, check-in frequencies, and tip suggestions to assess customer engagement and identify peak operational windows. These insights, generated through scalable Spark transformations, help businesses optimize staffing, promotions, and overall customer experience.

Key Features:

- **Data Ingestion and Storage**: Raw Yelp JSON files are downloaded and uploaded to Azure Data Lake Storage Gen2, creating a centralized and secure storage solution for structured and unstructured data.
- **Pipeline Orchestration**: Azure Data Factory's Copy Data pipeline automates the movement of data between raw and structured storage containers, ensuring a clean, modular data flow.
- **Scalable Data Processing**: Azure Databricks handles large-scale data transformations using PySpark, making the pipeline efficient, reliable, and easy to scale across increasing datasets.
- **Insight Generation**: Processed and curated data is made available for downstream analytics, enabling SQL-based querying and visualization in the Azure cloud environment.

As new business and review data is generated, it is seamlessly integrated into the pipeline, ensuring up-to-date insights on customer behavior and market trends. This real-time capability is crucial for dynamic industries like food, retail, and services, enabling Yelp-listed businesses to continuously adapt and grow based on reliable, data-driven feedback.

Use-case depicted in this project

This project implements a cloud-native data analytics pipeline using **Azure services** to automate the ingestion, structuring, and transformation of Yelp's business ecosystem data. The system is designed for high scalability, modularity, and real-time insights generation with minimal manual intervention. It leverages **Azure Data Lake Storage Gen2, Azure Data Factory, and Azure Databricks** to process large volumes of JSON-formatted Yelp data for business intelligence and customer experience analysis.

A real-world example would be a **market research agency** or **enterprise brand team** analyzing Yelp data to identify trends in customer sentiment, business category performance, and consumer behavior across geographies. The structured insights allow businesses to refine marketing strategies, optimize locations, and enhance customer satisfaction.

Workflow

This project builds a robust, end-to-end data pipeline on Azure to transform and analyze large-scale, semi-structured Yelp data. The solution is tailored for extracting business and consumer insights from JSON files and supports downstream use by analytics and data science teams.

1. Data Ingestion & Storage

Objective: Load raw Yelp JSON files into a secure, scalable cloud repository.

- **Source System:** Yelp public dataset containing business, review, user, check-in, and tip data in JSON format.
- **Ingestion:** The raw files are downloaded locally and uploaded to a **designated Azure Data Lake Storage Gen2 (ADLS)** container.

- **Storage Layer:** ADLS serves as the foundation for both raw and structured data zones, ensuring security and cost-effective long-term storage.

2. Data Structuring with Azure Data Factory

Objective: Orchestrate and automate the movement of raw data for processing.

- **Data Pipeline:** A **Copy Data pipeline** in **Azure Data Factory (ADF)** is configured to move files from the raw container to a structured container in ADLS.
- **Automation:** ADF ensures data is regularly and reliably copied, setting up a ready-to-process layer for downstream analytics.

3. Data Processing & Transformation with Azure Databricks

Objective: Clean, enrich, and analyze Yelp data for actionable insights.

- **Triggering & Computing:** Azure Databricks notebooks are used to run PySpark transformations on the structured data.
- **Transformations Include:**
    - Parsing and flattening nested JSON structures.
    - Cleaning null or irrelevant entries.
    - Merging business data with reviews, user activity, check-ins, and tips.
    - Generating analytical features (e.g., average star ratings per category, user engagement scores, peak visit times).

- **Insight Generation:**
    - Identifying **top-rated business categories by city or state**.
    - Detecting **customer sentiment trends** using NLP on reviews.
    - Analyzing **user loyalty and influence** based on review history and social connections.
    - Spotting **location-based performance patterns** using check-in and review density.

4. Storage & Analytics Integration

Objective: Store transformed data for dashboarding and model training.

- **Storage:** Final output is saved in a structured format for efficient querying and downstream analytics.
- **Analytics Ready:** Data is made available and can be used for further tasks:
    - **Power BI dashboards** showcase business intelligence insights.
    - **Machine learning models** are used to predict customer satisfaction or business success.
    - **Marketing strategy formulation** by identifying high-potential customer segments and business types.

Data Description

This project involves two datasets - Dataset 1 and Dataset 2, which are the Yelp dataset provided in JSON format, containing rich information about businesses, user interactions, reviews, and customer behavior.

The Yelp dataset offers a comprehensive, community-driven snapshot of consumer interactions with local businesses across various sectors. It is designed to enable data scientists and analysts to extract meaningful insights about business performance, customer behavior, and regional market dynamics.

The dataset is provided in JSON format and is composed of the following key components **(This is just a basic detail about the different Yelp JSON data, and not restricted to these attributes only):**

- **business.json**
  Contains detailed information about each business listed on Yelp, including:
  - Business name, address, and geolocation (city, state, latitude, longitude)
  - Categories (e.g., restaurant, retail, services)
  - Operational status (open or closed)
  - Business attributes (e.g., parking availability, Wi-Fi, payment methods)
- **review.json**
  Stores customer reviews and ratings for businesses, which include:
  - Review text and star rating (1 to 5)
  - Timestamp of the review
  - User ID and business ID for reference

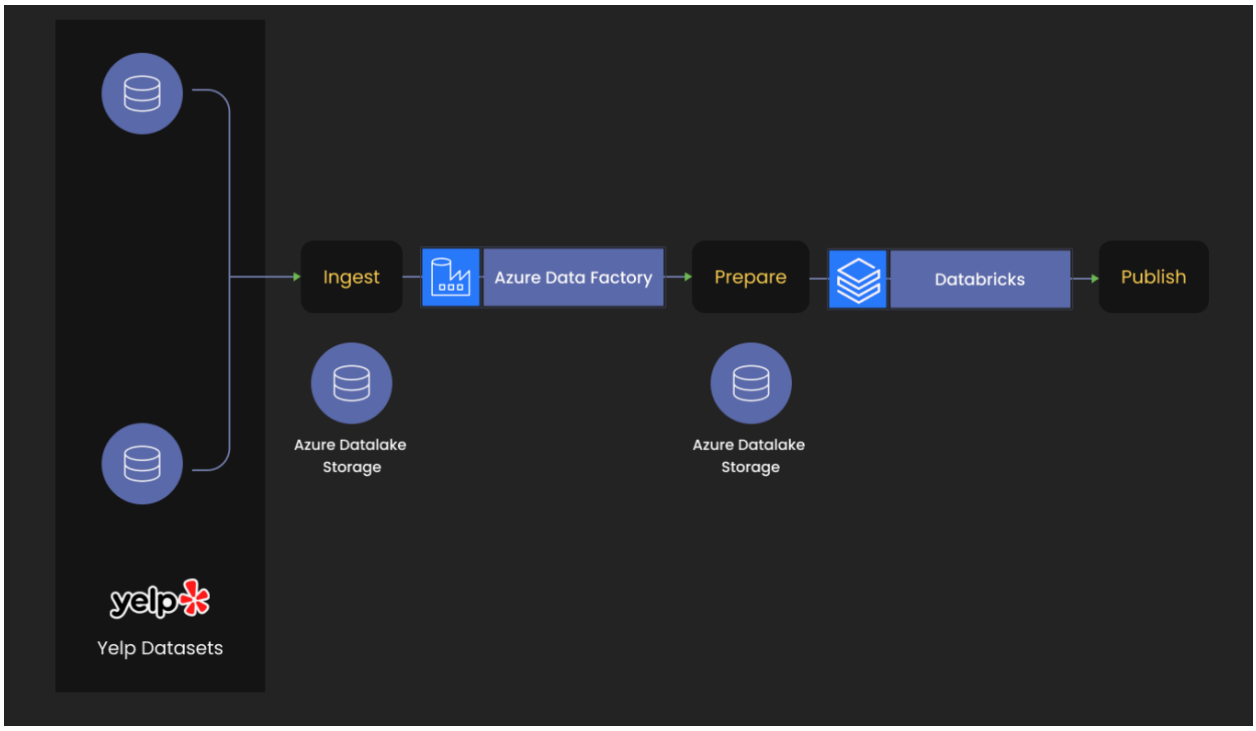
- Review usefulness, funny, and cool votes (user engagement metrics)

- **user.json**
  Captures metadata about Yelp users, such as:
  - User profile information (ID, name, joining date)
  - Review and tip activity counts
  - Social connections (friendship network)
  - User status indicators (elite years, compliments received)

- **checkin.json**
  Reflects business foot traffic patterns by logging:
  - Time and frequency of customer check-ins
  - Aggregated data on when customers are most active (hourly/daily trends)

- **tip.json**
  Represents short-form customer feedback, including:
  - Tips or suggestions for businesses
  - User IDs and timestamps for context
  - Optional likes or endorsements from other users

Architecture Diagram:



Prerequisites

Breakdown:

- **Setup required:**
  - Create an Azure account
  - Download the dataset - 1
  - Download the dataset - 2
  - Download the Code
  - This is an OS-independent project, as the whole pipeline is set up on the Azure Cloud
- **Ways to interact with the Azure services:**
  - Console- Using the Web-based UI
  - CLI- Using the Azure CLI tool in Terminal/CMD
  - For this project, we will be using the UI Console of Azure to work with different Azure Services
- **Best Practices for Azure Account**
  - Enable Multi-Factor Authentication (MFA)
  - Protect the account by adding only authorized users/ groups
  - Choose the region where the services work
  - Create a Budget and set up an alert notification
  - Keep track of Unused ongoing Services
  - Delete all the services after execution to cut down on charges.
  - If you use a free tier account, ensure all project services are available under the free tier; upgrade if needed.
  - Rotate all keys and passwords periodically
  - Follow the least privilege principle
  - Follow a naming convention for resources for better understanding, and name them after the project's aim.

Services

Azure Storage Account:

Azure Storage Account is a fundamental service in Azure that provides highly scalable, durable, and secure cloud storage for a variety of data types. It serves as the container for storing data in different formats, such as blobs, files, queues, and tables. There are several types of storage accounts in Azure, each tailored to specific needs, such as Blob Storage and Azure Data Lake Storage Gen2.

a) **Blob Storage Account**:
   - **Purpose**: The storage account comes under the Azure Storage account service, which is designed for storing large amounts of unstructured data, such as text or binary data. It's typically used for storing files like images, videos, documents, backups, and logs. In our project, the container is created inside the blob storage account to store the JSON data coming from the execution of Logic Apps.

b) Azure Data Lake Storage Gen22 (ADLS):
   - **Purpose: It is** built on top of Azure Blob Storage, but Data Lake Storage Gen2 is optimized for big data analytics workloads and provides more advanced features for storing and managing data at scale. In our project, ADLS acts as a data source in Azure Databricks during the execution of Spark jobs.

Advantages of Azure Storage Account:

- With the ability to scale horizontally across both Blob Storage and Data Lake Storage Gen2, the system can handle increasing data loads from both unstructured raw data
- Using Blob Storage for frequently accessed raw data and Data Lake Storage Gen2 for large-scale data processing ensures we pay only for the storage you need while maximizing cost efficiency.

- Both storage types provide a robust set of security features, but Data Lake Storage Gen2 offers more granular control over who can access specific parts of the data.
- By using both storage accounts, it is ensured that raw data in Blob Storage is highly available and resilient to failures
- Raw JSON data stored in Blob Storage can be easily moved to Data Lake Storage Gen2, where it can be processed using Spark jobs, machine learning models

Alternatives:

- Amazon S3 (Simple Storage Service)
- Google Cloud Storage
- IBM Cloud Object Storage

Azure Data Factory(ADF):

Azure Data Factory (ADF) is a cloud-based data integration service provided by Microsoft Azure. It allows you to create, schedule, and orchestrate data workflows. ADF helps move and transform data from one location to another, using a visual interface or a code-based approach. In this project, **Azure Data Factory (ADF)** will be used to orchestrate the movement of raw data stored in **Azure Blob Storage** to **Azure Data Lake Storage Gen2 (ADLS)**. ADF's **Copy Pipeline will be used**, and the JSON data in Blob Storage will be transferred to ADLS.

Advantages of Azure Data Factory:

- **Scalability:** ADF is highly scalable, making it suitable for both small and large-scale data integration projects.
- **Integration with Azure Services**: ADF integrates seamlessly with other Azure services, such as Azure Synapse Analytics, Azure Data Lake Storage, and Azure Machine Learning.
- **Cost-Effective**: ADF offers a pay-as-you-go pricing model, which means you only pay for the resources you use.
- **Security**: ADF provides robust security features, including data encryption, network isolation
- **Ease of Use**: The intuitive visual interface of ADF allows users to create and manage data pipelines without extensive coding knowledge.

Alternatives:

- AWS Glue
- dbt
- Apache NiFi

Azure Databricks:

Azure Databricks, developed in collaboration with Microsoft, is a managed version of Databricks that allows Azure customers to set up with a single click, streamline workflows, and access shared collaborative interactive workspaces. It facilitates rapid collaboration among data scientists, data engineers, and business analysts through the Databricks platform. Azure Databricks is closely integrated with Azure storage and compute resources, such as Azure Blob Storage, Data Lake Store, SQL Data Warehouse, and HDInsights.

Advantages of Azure Databricks:

- While Azure Databricks is Spark-based, it allows commonly used programming languages like Python, R, and SQL to be used. These languages are converted in the backend through APIs, to interact with Spark.
- Deploying work from Notebooks into production can be done almost instantly by just tweaking the data sources and output directories.
- Aside from those Azure-based sources mentioned, Databricks easily connects to sources, including on-premise SQL servers, CSVs, and JSONs.
- While Azure Databricks is ideal for massive jobs, it can also be used for smaller-scale jobs, deployment, and development/ testing work.
- Databricks notebooks allow for real-time collaboration among data engineers, data scientists, and analysts using version control.

Alternatives:

- Amazon EMR (Elastic MapReduce)
- Google Cloud Dataproc
- Apache Spark on Kubernetes

Partition and Coalesce

In this project, you're working with large files like **review.json, business.json,** etc., stored in JSON format. When you load them into a Spark DataFrame in Databricks, Spark automatically creates partitions **(the default is 200).**

But... this automatic partitioning is not always efficient.

So, understanding repartition() and coalesce() is critical to:

- Optimize parallel processing (like filtering, joining, and aggregating)
- Avoid small file problems when writing data back to storage

Before jumping into each (Partition and Coalesce), you need to understand the logical steps Spark takes during a transformation:

1. Logical Plan: Spark builds a blueprint of what needs to be done.
2. Physical Plan: Spark converts this into a physical plan based on transformations.
3. RDD Transformation: Every DataFrame operation is translated into low-level RDD operations.
4. Tasks: Each RDD partition becomes a task sent to an executor.
5. Shuffling (if needed): Spark redistributes data between partitions/nodes during wide transformations.

Partitioning:

Data partitioning in Spark means dividing a large dataset into smaller, manageable chunks called partitions. Each partition is processed in parallel by Spark executors to speed up computation.

Why does it matter?

Good partitioning helps Spark:

- Use cluster resources efficiently
- Avoid data skew and bottlenecks
- Optimize parallel processing

What is repartition()?

The Repartition () is a transformation in Spark used to increase or change the number of partitions in a DataFrame by reshuffling the entire dataset across the network. It is a wide transformation, meaning it requires a full shuffle of the data, which can be expensive. It is used when you want to better balance data before heavy operations like groupBy, join, or to improve parallelism.

Practical Example:

Imagine you have a Yelp review dataset with just 4 partitions, and you are working on a 16-core cluster. You want Spark to use more cores efficiently.

```
# Load data

df = spark.read.json("path/to/review.json")


# Increase number of partitions to 16

df_repartitioned = df.repartition(16)


# Check partitions

print("Partitions after repartition:",
df_repartitioned.rdd.getNumPartitions())
```

When to use:

- Before joins or aggregations
- When the original partitions are too big
- To rebalance uneven partitions

What happens under the hood:

Spark shuffles all data, which can be slow, but balances load across workers.

1. **Conversion to RDDs**: DataFrame becomes RDD (or uses underlying RDD).
2. **Hash Partitioning**: Spark uses HashPartitioner or similar logic to determine which partition a row belongs to.

3. **Shuffle Write Phase**:
    - All existing partitions' data is **written to temporary files on local disks**.
    - Each row is assigned to a new partition ID (0 to n-1).
4. **Shuffle Read Phase**:
    - New tasks **read the partition data** from multiple sources over the network.
    - The output is now evenly distributed across the n partitions.
5. **A new RDD lineage** is created with n partitions.

What is coalesce()?

The Coalesce () is used in Spark to reduce the number of partitions in a DataFrame without a full shuffle. It is an efficient transformation for reducing partition count, especially when writing data to disk. Unlike repartition(), coalesce() simply merges adjacent partitions and is ideal for performance optimization at the output stage.

Practical Example:

After processing the Yelp data, you want to write the result into a single output file.

```
# Reduce the number of partitions to 1

df_single = df.coalesce(1)
```

```
# Write to disk

df_single.write.mode("overwrite").parquet("path/to/output/"
)
```

When to use:

- Before writing data, to avoid small files
- When decreasing the number of partitions count
- When a shuffle is not needed

What happens under the hood:
Spark does not shuffle; it merges partitions, which is faster.

1. **Partition Mapping**:
    - Spark selects a subset of existing partitions and **combines them logically**.
    - Data is **not moved between executors**.
2. **No Shuffle**:
    - Instead of moving data, Spark just **tells fewer tasks to read more data** from existing partitions.
    - Spark **avoids a new shuffle stage** in the DAG.

3. **Task Execution**:
    - Each new partition is formed by **reading data from one or more adjacent partitions**.
    - No repartitioning logic, just fewer tasks handling more input data.

Why it's efficient:

- **No disk/network overhead** (only logical grouping).
- Very fast for operations like coalesce(1) to write a single file.

Scenario

You're working with the **review.json file** from the Yelp dataset, which contains millions of customer reviews. This dataset is loaded into a Spark DataFrame (df) and currently has **8 partitions**.

Your Spark cluster consists of **4 worker nodes**, and each node is handling **2 partitions**, which gives a total of 8.

Now, your goal is to either:

- Optimize the dataset for **parallel processing** before a groupBy operation, or
- Reduce the number of partitions before writing it to disk to avoid creating too many small output files.

Question:

The **Yelp review.json dataset** is currently split into 8 partitions across 4 worker nodes (2 partitions per node).

If you apply:

1. df.repartition(4) and
2. df.coalesce(4)

What will be the backend behavior in each case? Which one is more efficient in this scenario, and why?"

Answer:

Case 1: df.repartition(4)

What Spark does:

- Spark **performs a full shuffle** of data across the cluster.
- It reads all 8 existing partitions, and **randomly redistributes** the data across 4 new partitions.
- Each partition is likely to be more **evenly balanced**, ideal for parallel processing.

Backend Process:

1. Spark creates **new tasks** for the repartitioned data.
2. All executors write temp shuffle files to disk.
3. Shuffle is triggered — data is moved between nodes over the network.

4. 4 new partitions are created with balanced records.

Cost:

- **High CPU**, **network I/O**, and **disk usage** due to shuffling.
- **Creates a new DAG stage** in the execution plan.

Best Use Case:

- Use only when you **want to increase partitions** or **prepare data for wide transformations** (like join, groupBy, etc.).

Case 2: df.coalesce(4)

What Spark does:

- Spark **avoids shuffling** and simply **merges adjacent partitions**.
- For example, it might just say: "Partition 0 and 1 → P0', Partition 2 and 3 → P1', etc."
- Data stays on the same nodes, reducing overhead.

Backend Process:

1. Spark maps the existing 8 partitions to 4 new logical groups.
2. Tasks are created to **read from multiple old partitions** without moving data.
3. **No shuffle**, **no disk writes**, **no network traffic**.

Cost:

- Very **low**, just merges metadata and reuses the existing data layout.

Best Use Case:

- Use when you're **reducing partitions**, such as before df.write(), to reduce the number of output files

Parquet File Format

In this project, we will learn how to read data from a container (e.g., Azure Blob Storage, AWS S3, or a local containerized environment) into a DataFrame using the JSON format, and then write that data back into the container in **Parquet** format. The reason we choose Parquet is due to its **superior read and write efficiency** and **built-in optimization techniques**, especially for large-scale data analytics.

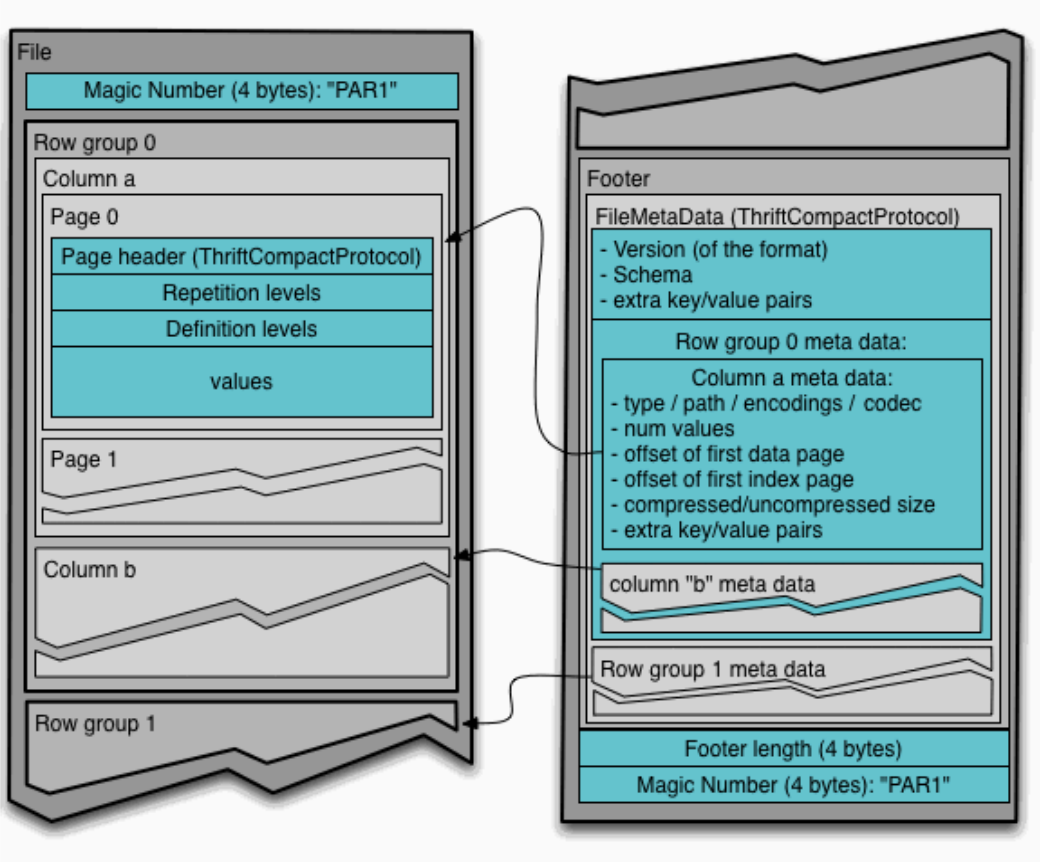
What is Parquet Format?

Parquet is a columnar storage file format designed for efficient data storage and retrieval. Developed as part of the Apache Hadoop ecosystem, it is optimized for analytical workloads that scan large volumes of data. Unlike row-based formats like CSV or JSON, Parquet stores data **column-wise**, which means it stores all the values of a column together rather than storing entire rows.

Internals of Parquet Format

Internally, a Parquet file is made up of:

- **File Header** – contains magic bytes for format identification.
- **Row Groups** – each row group contains a set of rows and consists of column chunks.
- **Column Chunks** – store the actual columnar data for each column in a row group.
- **Page** – the smallest unit of data storage, which can be dictionary-encoded or data pages.
- **Footer** – contains metadata such as schema, column statistics, and file offsets for fast access.



Parquet supports **schema evolution**, **predicate pushdown**, **column pruning**, and **compression** (like Snappy, GZIP), which allows for reduced disk I/O and faster query execution.

Internal Working of Parquet File Format

1. Reading JSON into DataFrame

Let's assume we are using **PySpark** to read a JSON file from a storage container:

df = spark.read.json("s3://my-bucket/input/data.json")

- Spark parses the JSON line by line.
- It infers or uses a provided **schema** to structure the data.
- Internally, Spark stores this as a **Row-based in-memory DataFrame**.

2. Writing DataFrame to Parquet Format

```
df.write.parquet("s3://my-bucket/output/data.parquet")
```

Internal Steps:

1. **Schema Conversion**:
    - Spark maps the DataFrame schema to the **Parquet schema** using the **Apache Arrow** in-memory format for performance.
2. **Data Partitioning (Optional)**:
    - If specified (e.g., `df.write.partitionBy("year")`), the data is **logically split** into folders based on column values.

3. **Columnar Conversion**:
    - Each column is stored **separately**.
    - Instead of storing one row at a time, Parquet stores **column chunks** inside **row groups**.
4. **Page Creation**:
    - Each column chunk is further divided into **pages**.
    - Pages may include **dictionary pages** (for repeated values) and **data pages**.
5. **Compression**:
    - Each column/page is optionally **compressed** (e.g., using Snappy).
    - Compression is more effective because values in a column are often similar (e.g., “CA”, “CA”, “NY”, “CA”).
6. **Metadata Writing**:
    - Parquet writes a **footer** containing the schema, column statistics (min, max, null count), and file offsets.
    - This allows **fast filtering** and **schema-aware reading**.

Reading Parquet Files (Internals)

```
df = spark.read.parquet("s3://my-bucket/output/data.parquet")
```

1. **Footer Read First**:
    - Spark reads the **footer** to understand the schema and file layout.
    - It uses statistics to apply **predicate pushdown**.
2. **Column Pruning**:
    - If your query only requests some columns, Spark will **only read those columns**.
    - This avoids full file scans.
3. **Predicate Pushdown**:
    - If a filter like `df.filter("state == 'CA'")` is used, Spark uses **column stats** to skip irrelevant row groups.
4. **Lazy Loading & Parallelism**:
    - Data is read **lazily** and only when needed.
    - Parquet supports **parallel reading**, which speeds up processing in distributed systems.

Example Scenario

Imagine a DataFrame with customer purchase data:

customer_id	country	amount	purchase_date
1	US	200	2024-01-01
2	IN	350	2024-01-02

If you run:

```
df.filter("country = 'IN'").select("amount").write.parquet("s3://.../filtered/")
```

Parquet's internals will:

- Use the footer to check which files/row groups have `country = 'IN'`.
- Only read the **`country` and `amount`** columns (column pruning).
- Skip all others using **predicate pushdown**.
- Write the result using **compressed, columnar format** for future efficiency.

Delta Format

Delta Lake is an open-source storage layer that brings ACID transactions, schema enforcement, and time travel to big data workloads on top of Parquet files. It's developed by Databricks and designed to work natively with Apache Spark.

While Parquet is a file format, Delta is a transactional layer on top of Parquet that adds advanced data management capabilities like updates, deletes, merges, versioning, and rollback.

Real-World Use Case:

Imagine a retail business storing daily sales data. With regular Parquet, it's hard to:

- Update rows if there's a return.
- Track versions for auditing.
- Handle concurrent writes safely.

With Delta, you can:

- Use `MERGE`, `UPDATE`, or `DELETE` operations.
- Keep a versioned history of the data.
- Use schema enforcement to prevent inconsistent data.

Internal Working of Delta Format (Step-by-Step)

Step 1: Write JSON to Delta Table

```
df = spark.read.json("s3://my-bucket/input/data.json")

df.write.format("delta").save("s3://my-bucket/output/delta-sales/")
```

What Happens Internally:

1. Schema is Inferred or Defined.
2. Data is saved as Parquet files, but not just that!
3. Delta Log Directory (_delta_log) is created:
    - Every write operation generates a JSON or Parquet log file.
    - This contains metadata, file actions, and schema info.
4. Transaction Log tracks changes over time, similar to a version control system (e.g., Git).

Step 2: Delta Table Metadata Management

The Delta table tracks:

- Additions and deletions of files (not entire rows).
- Schema changes and evolution.

- File stats, such as min/max values for columns, null count, etc.

Step 3: Read from the Delta Table

```
df =
spark.read.format("delta").load("s3://my-bucket/output/delta-sal
es/")
```

Internally:

- Delta reads the transaction log first (_delta_log/00000000000000000001.json).
- Builds the latest snapshot of the table.
- Parquet files corresponding to the current table version are read.
- If filters are applied, predicate pushdown and column pruning are still used via Parquet.

Step 4: Update or Merge

```
from delta.tables import DeltaTable

delta_table = DeltaTable.forPath(spark,
"s3://my-bucket/output/delta-sales/")
```

```
delta_table.update(
  condition="customer_id = 1",
  set={"amount": "amount + 100"}
)
```

Internals:

1. Spark identifies which Parquet files contain `customer_id = 1`.
2. Those files are rewritten with updated rows.
3. A new log entry is added in `_delta_log` (e.g., version 2).
4. Old files are retained for time travel (unless vacuumed).

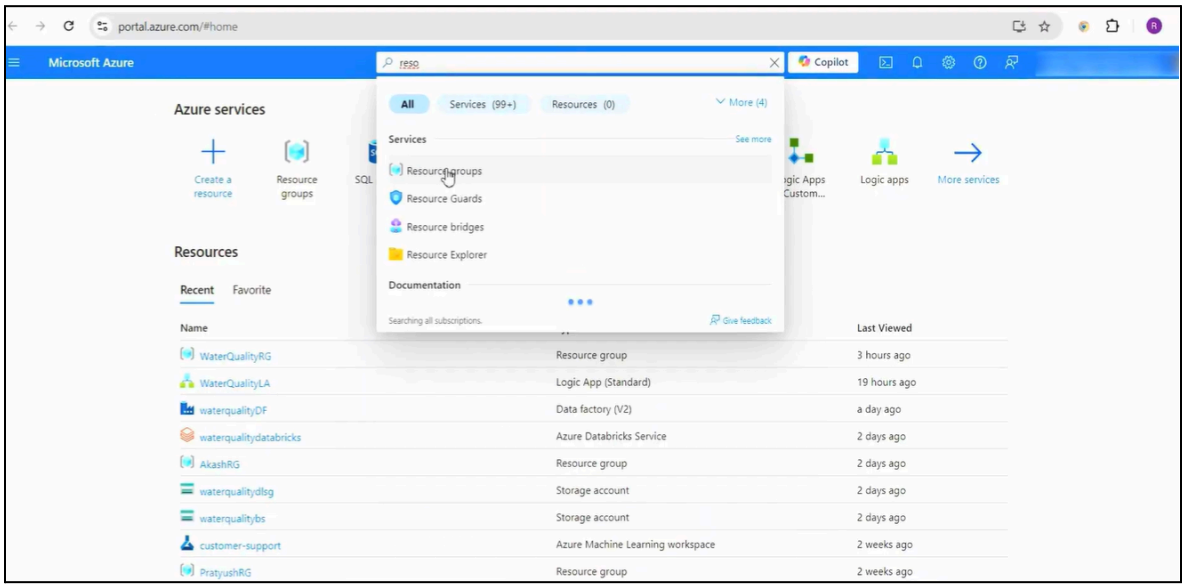
Delta Lake is optimized by combining **Parquet's columnar storage** with a transaction log that enables ACID compliance, fast updates/deletes, time travel, data skipping, and efficient schema management, all leading to faster, reliable, and scalable data processing.

Execution

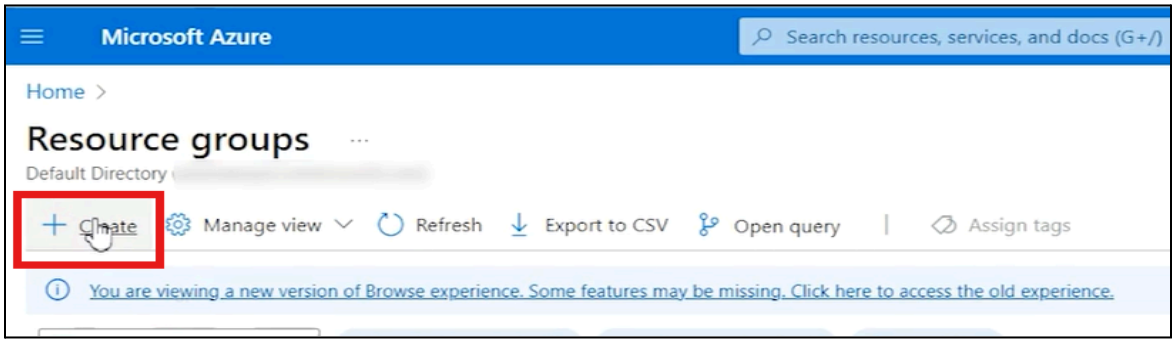
Creating Azure Resource Group

A Resource Group is created in Azure to manage all Azure Services effectively in one place. It is easier to monitor, update, or delete all Azure resources together as a single unit.

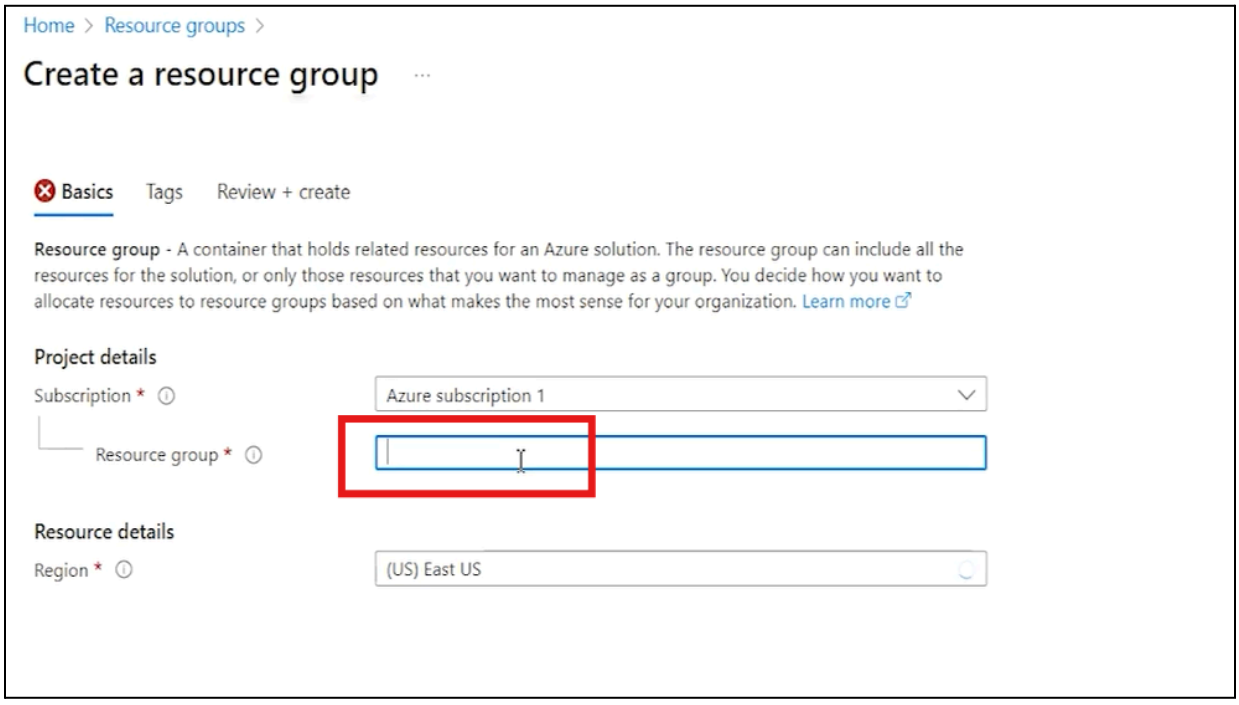
a) In the search bar, enter **"Resource Groups"** and select it from the suggestions.



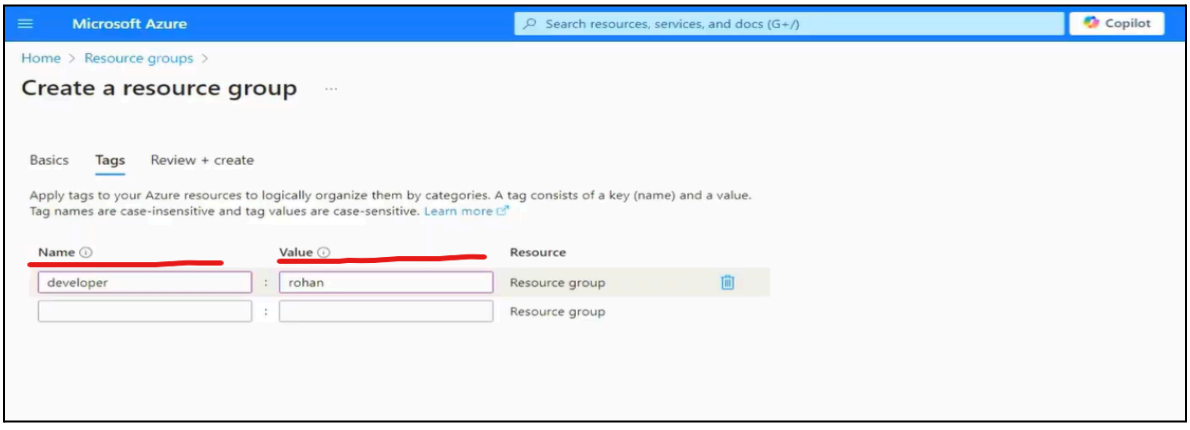
b) In the top-left corner, click on **Create**.



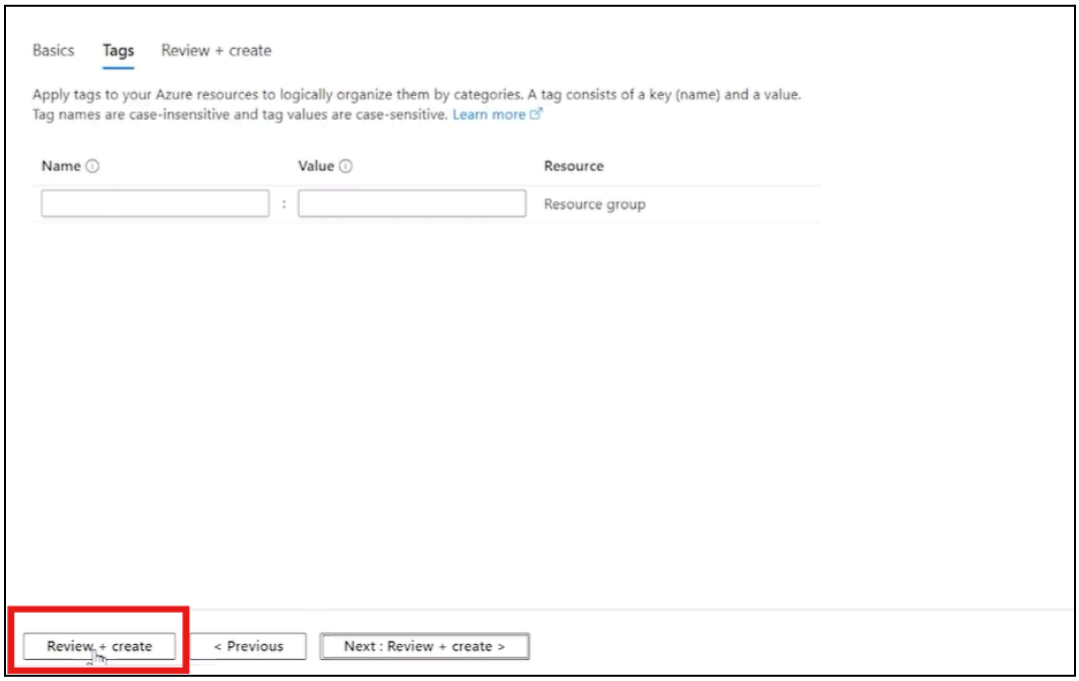
c) In the **Resource Group Name** field, enter the desired name for your resource group. Then, click on **Next: Tags**

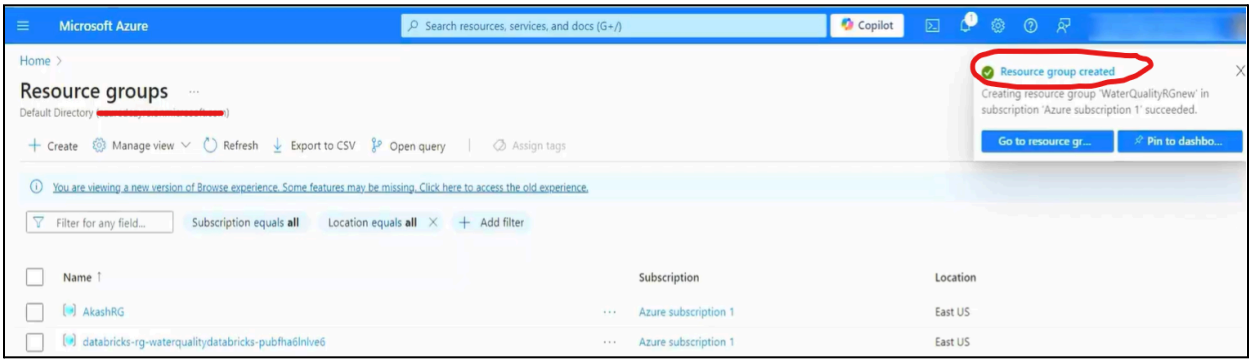


d) Under **Tags**, in the **Name** field, enter something like *developer.* In the **Value** field, enter your name.



e) Then, click on the **Create** option, and a resource group will be created.





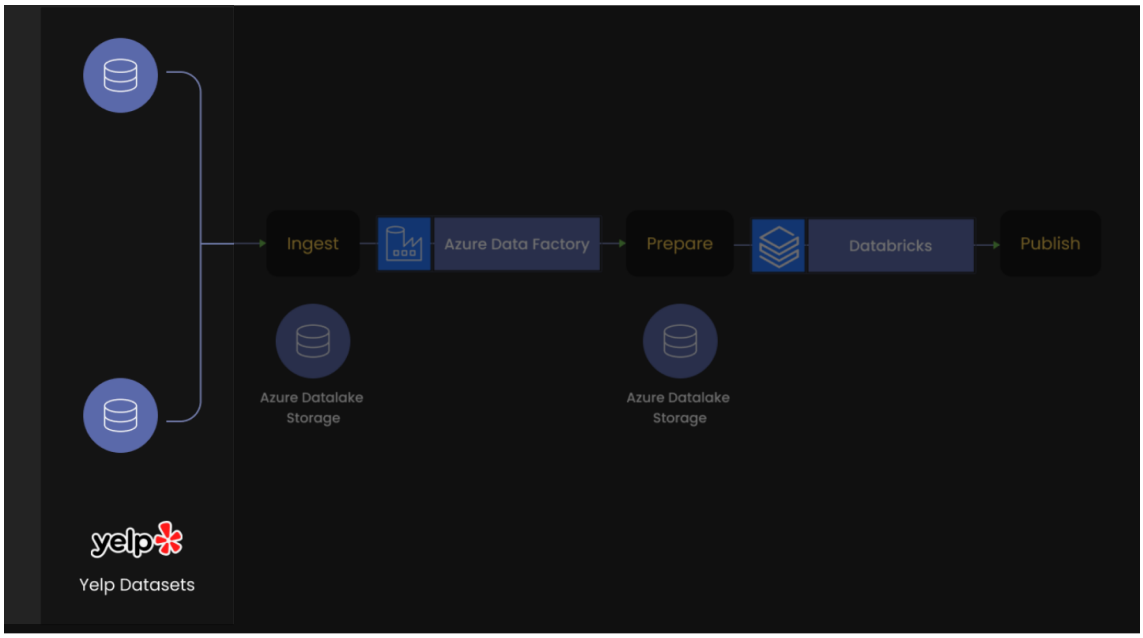
Creating an Azure Data Lake Storage Account (ADLS Gen2) and its container

After creating the resource group, the first step in the project is to create a storage account **using Azure Data Lake Storage Gen2 (ADLS Gen2).** This storage account will serve as the foundation for managing all project-related data. Inside this account, two containers will be created:

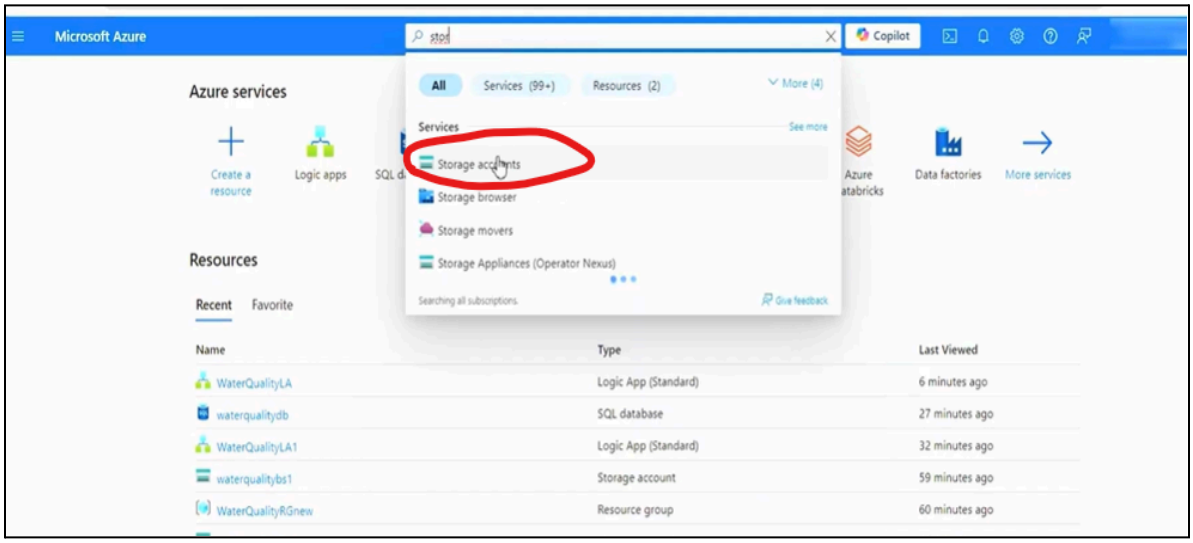
- The first container will store the **raw Yelp JSON files**. It acts as the source of truth, holding unprocessed data in its original form.

- The second container will receive a copy of this data through a copy pipeline that will be built using Azure Data Factory (ADF). This pipeline ensures that data is consistently transferred from the first container to the second.

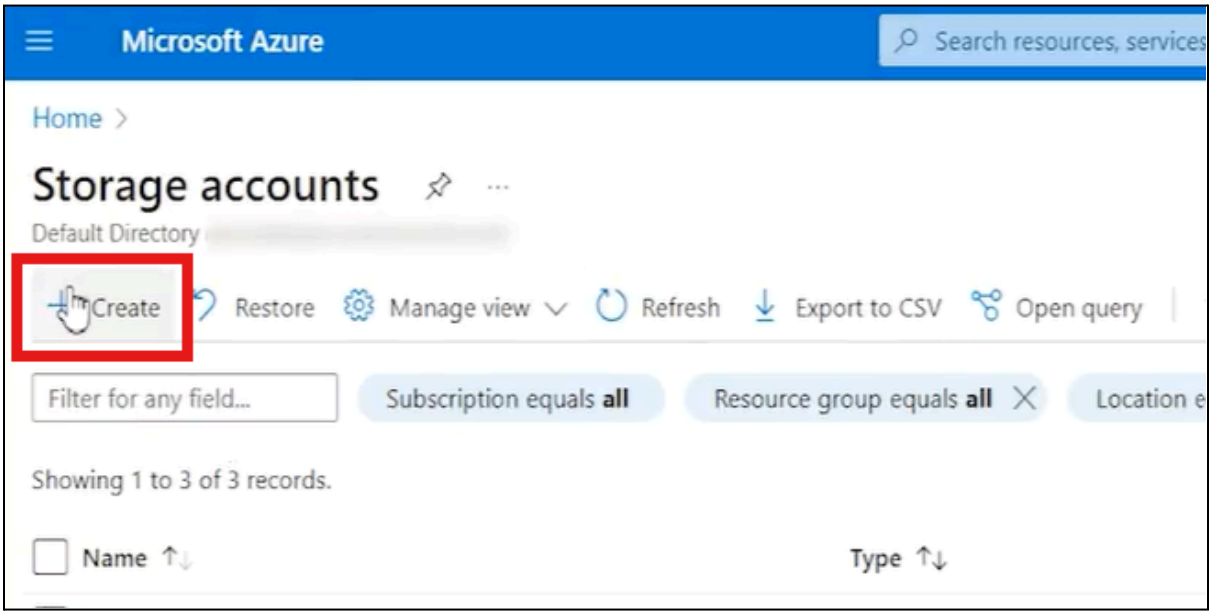
Once this setup is complete, the second container becomes the source from which Databricks will read the **files into DataFrames for transformation and analysis.** This structure enables a clean separation of raw and processing-ready data, supporting scalability and maintainability in the overall pipeline.



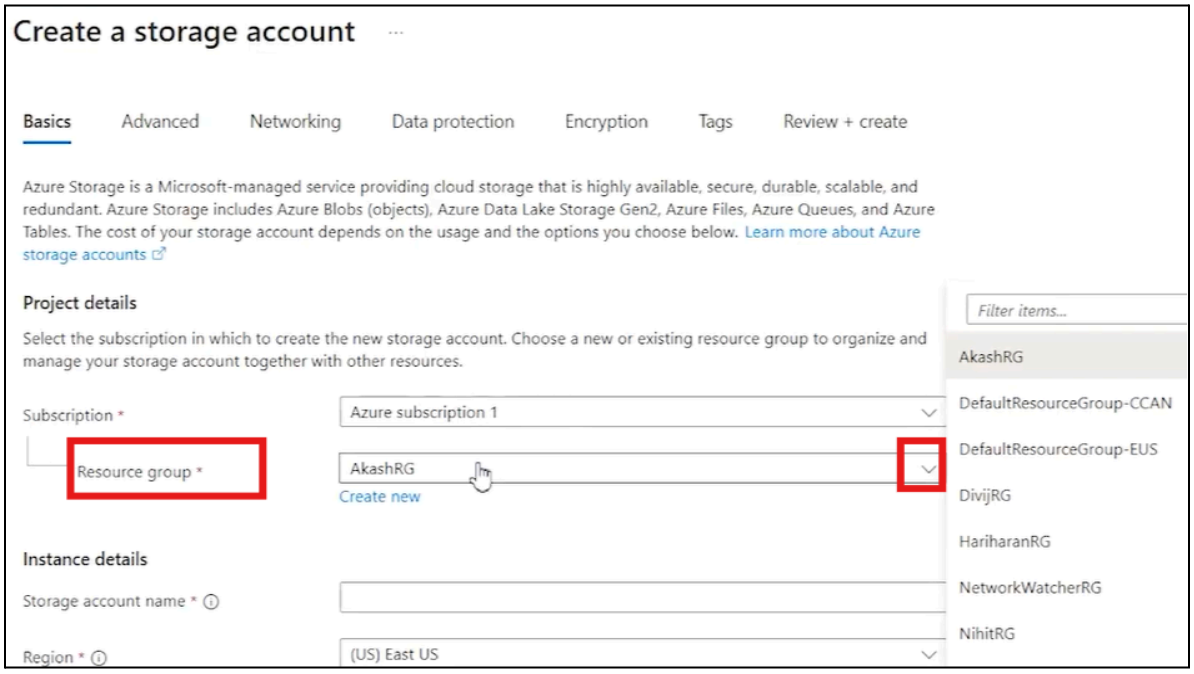
a) Navigate to the Azure home portal, type **Azure Storage Accounts** in the search bar, and select **Storage Accounts** from the suggestions.



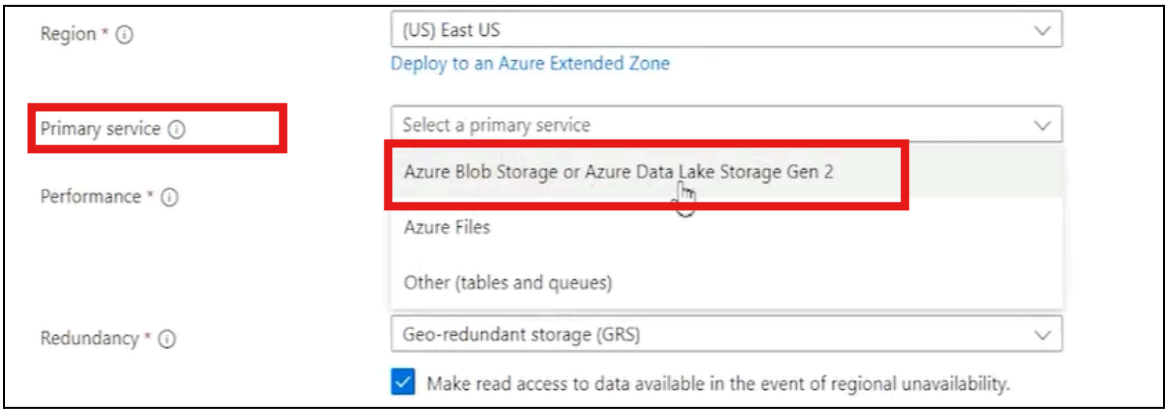
b) Click on **Create**



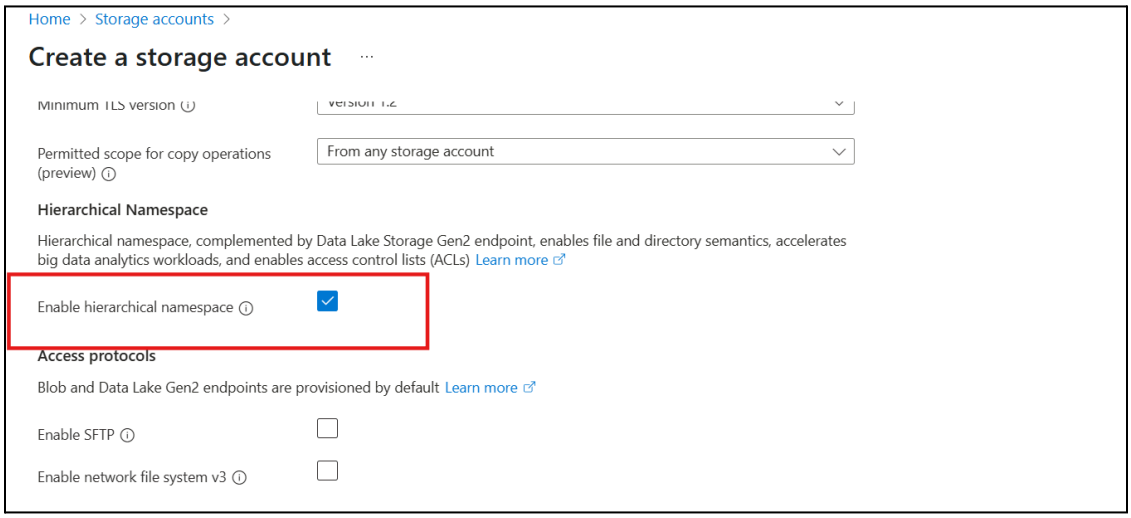
c) In the resource group field. From the dropdown, select the **resource group** you created and also specify the name of your ADLS Storage account as you want, and it should be unique.



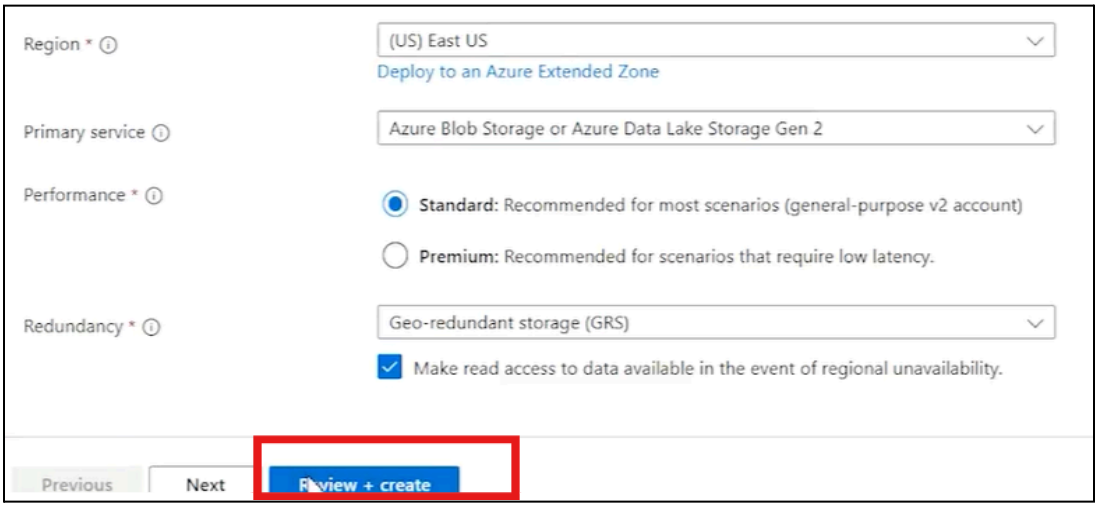
d) In the primary service, from the dropdown - choose **the "Azure Blob Storage or Azure Data Lake Storage"** option.



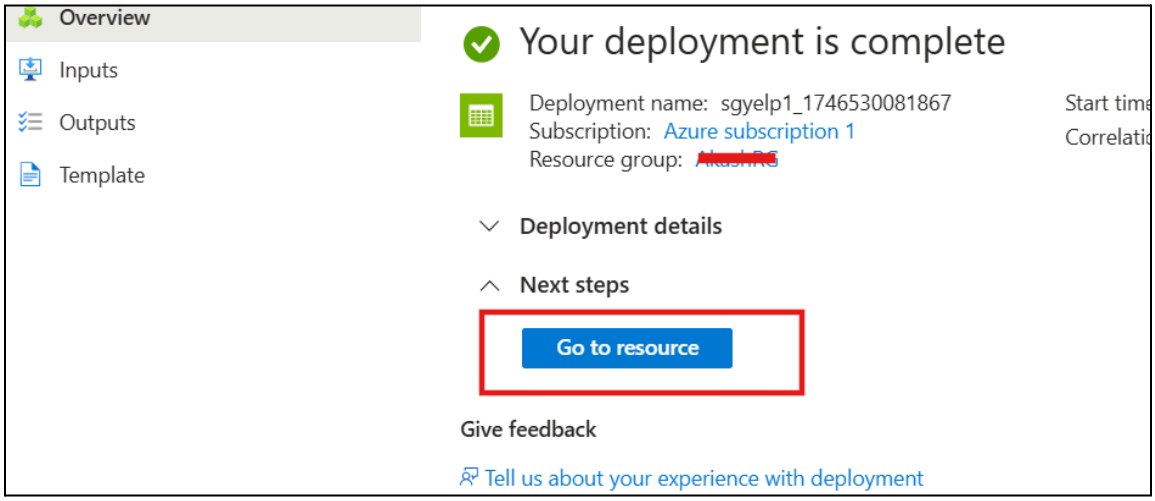
e) Click on next and in the advanced tab, enable **hierarchical namespace** to create Azure Data Lake Storage Gen2.



f) Then click on "**Review+Create" and** then click on **"Create" to create an ADLS Gen2 Account,** and then wait until the validation is done.



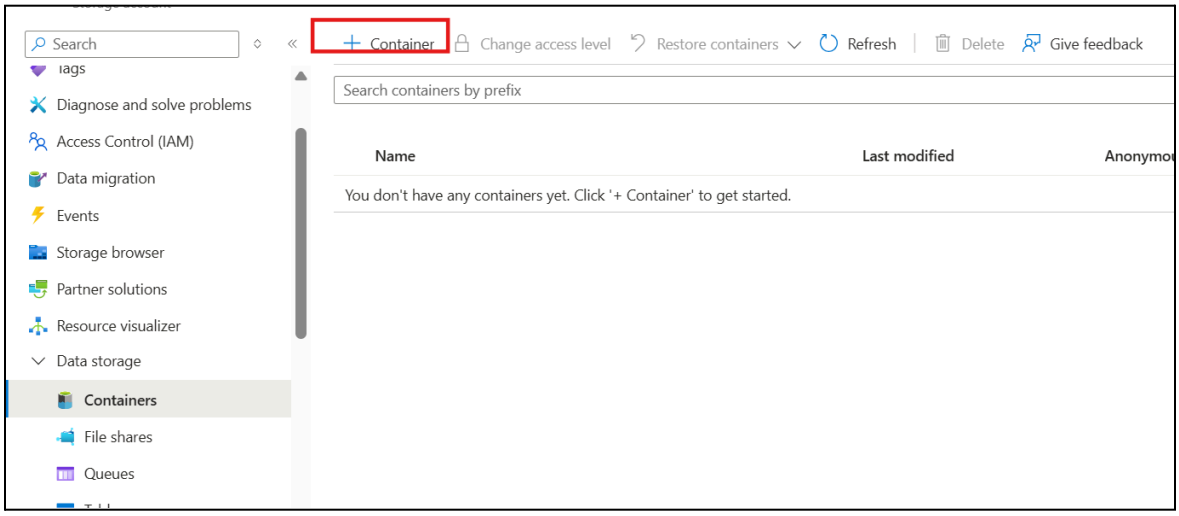
g) Once the deployment of the ADLS account is done, click on **“Go to Resource”.**

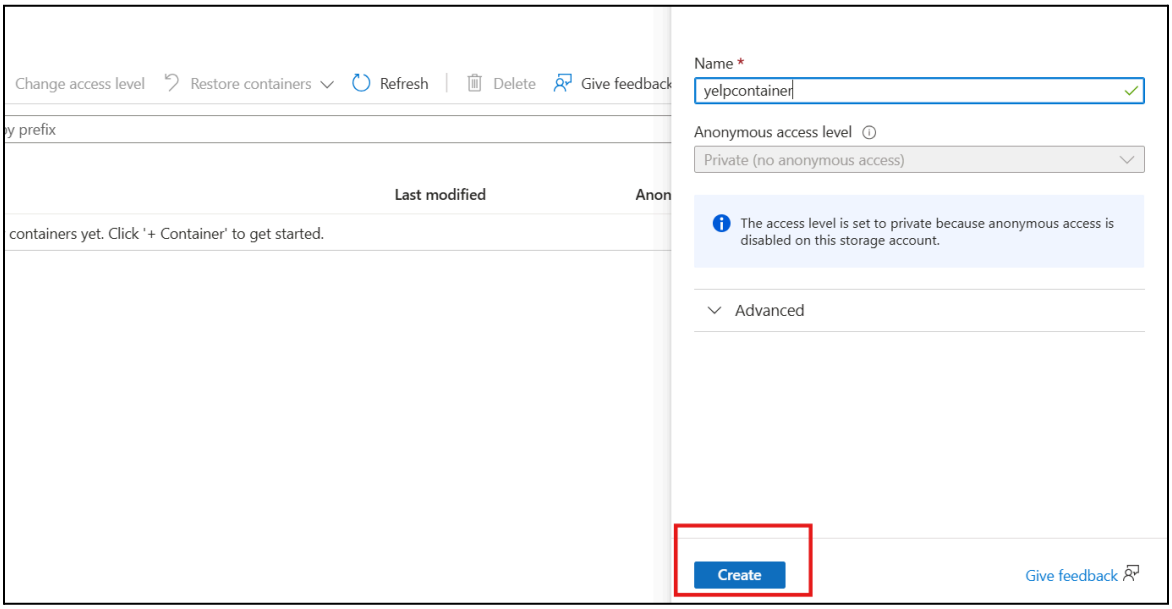


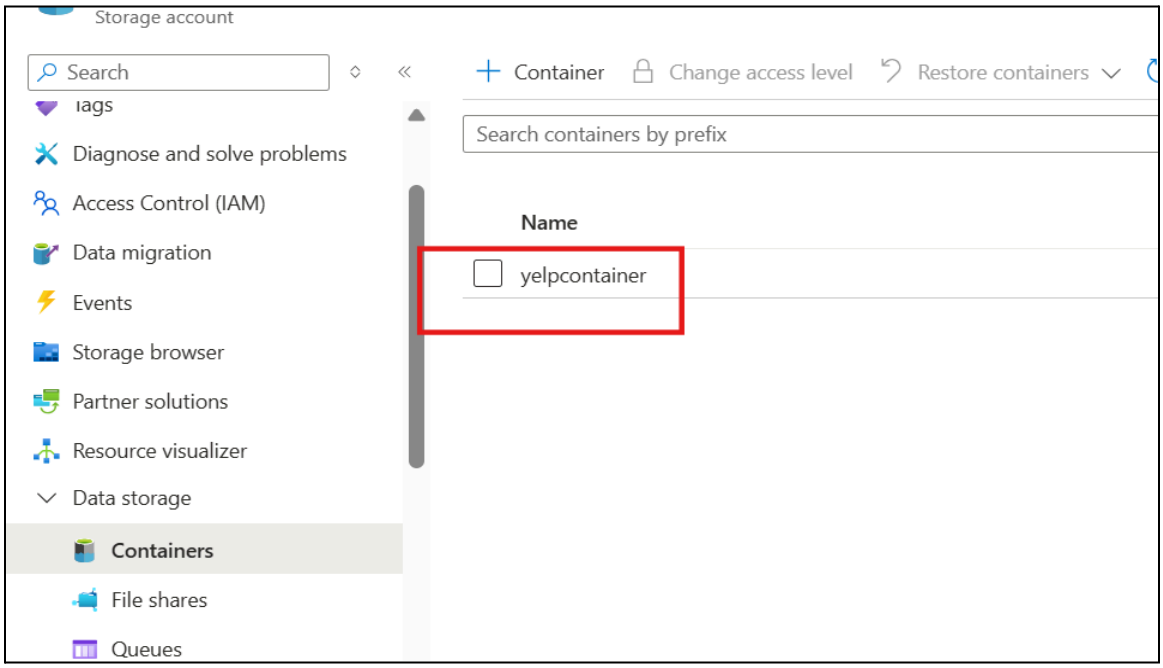
h) In the left-side panel, click on **Data Storage**, and under **Storage**, click on **Containers**.

i) Click on **+Container**. In the right-side panel, specify the name of the ADLS container as desired, then click on **Create**.

1. Firstly, we will create a container **(yelpcontainer)** that will store the raw Yelp JSON Data files.

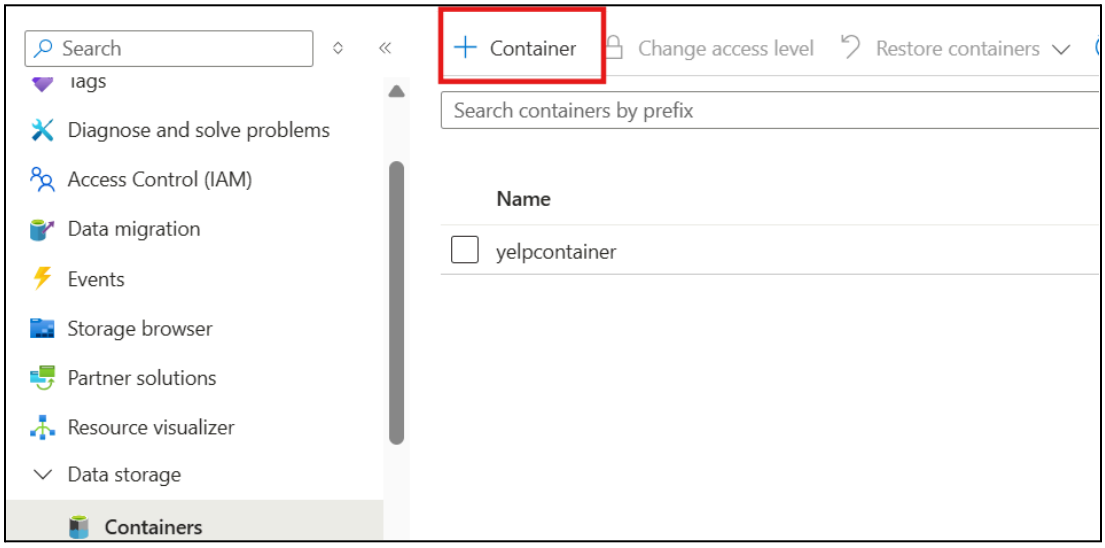


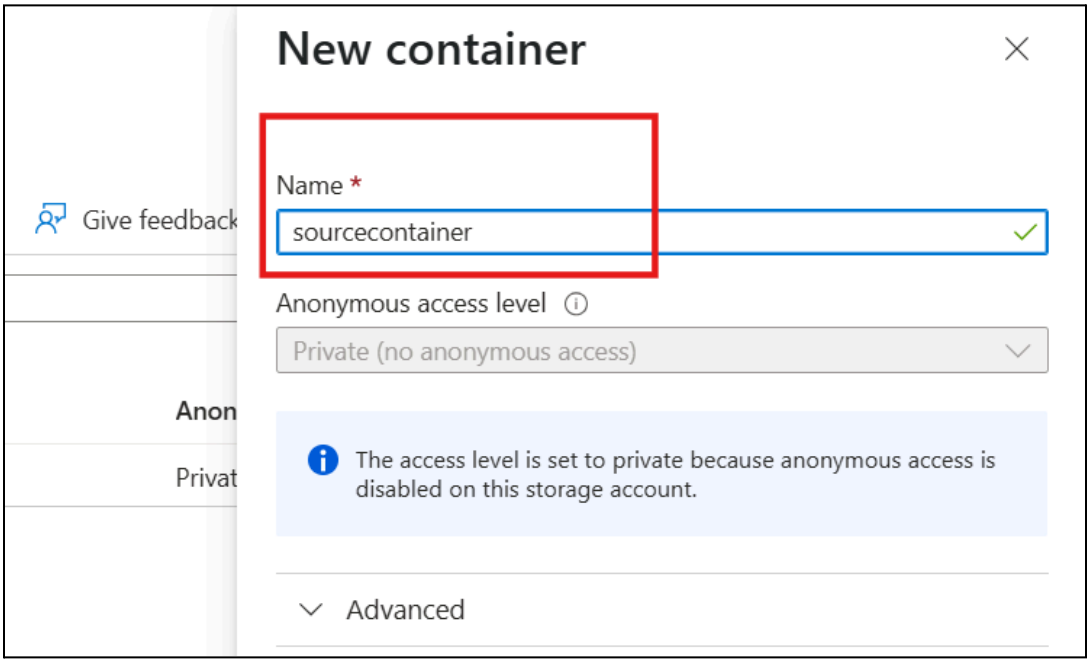


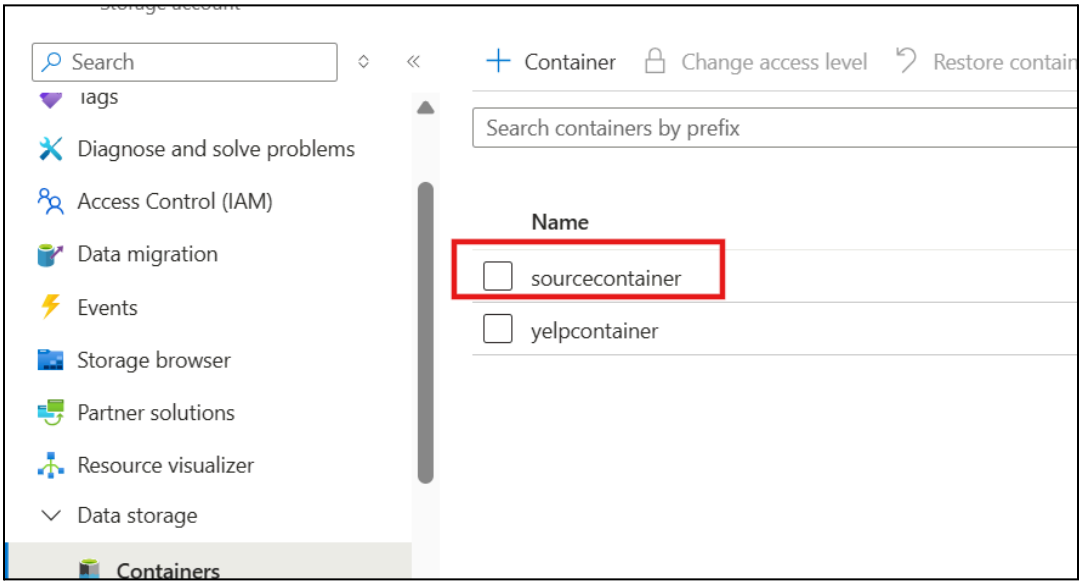


2. Now, we will create the second container (**sourcecontainer**) within the ADLS Gen2 storage account.

   This container is intended to store the Yelp data files that will be transferred from the first container using the **copy pipeline in Azure Data Factory (ADF)**, which will be configured and executed later in the project.


Search
Tags
Diagnose and solve problems
Access Control (IAM)
Data migration
Events
Storage browser
Partner solutions
Resource visualizer
Data storage
Containers
Container
Change access level
Restore containers
Search containers by prefix
Name
yelpcontainer


New container
Give feedback
Name *
sourcecontainer
Anonymous access level
Private (no anonymous access)
Anon
Privat
The access level is set to private because anonymous access is disabled on this storage account.
Advanced



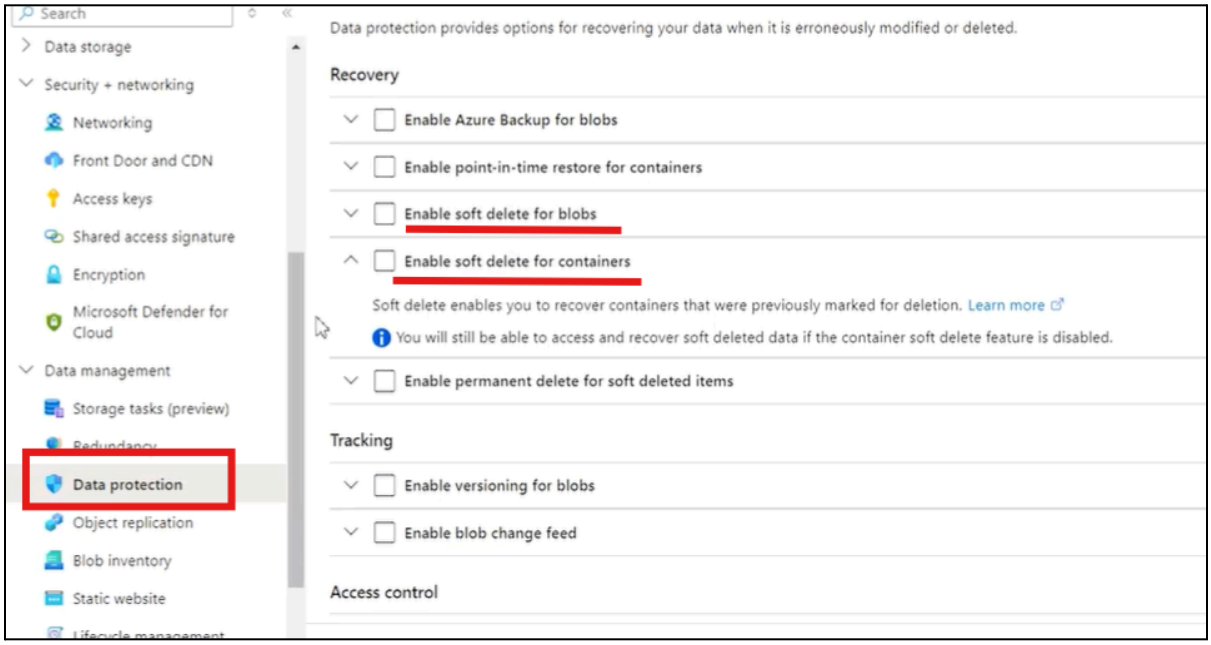
j) Navigate to the **ADLS Storage Account (not storage container)** and in the left-side panel of the ADLS Storage Account, scroll down and click on **Data Management**. Under the **Data Management** option, click on **Data Protection**.

Once you find the **Data Protection** option, in the right-hand side panel, uncheck the two options:

1. **Enable soft delete for blobs**
2. **Enable soft delete for containers**

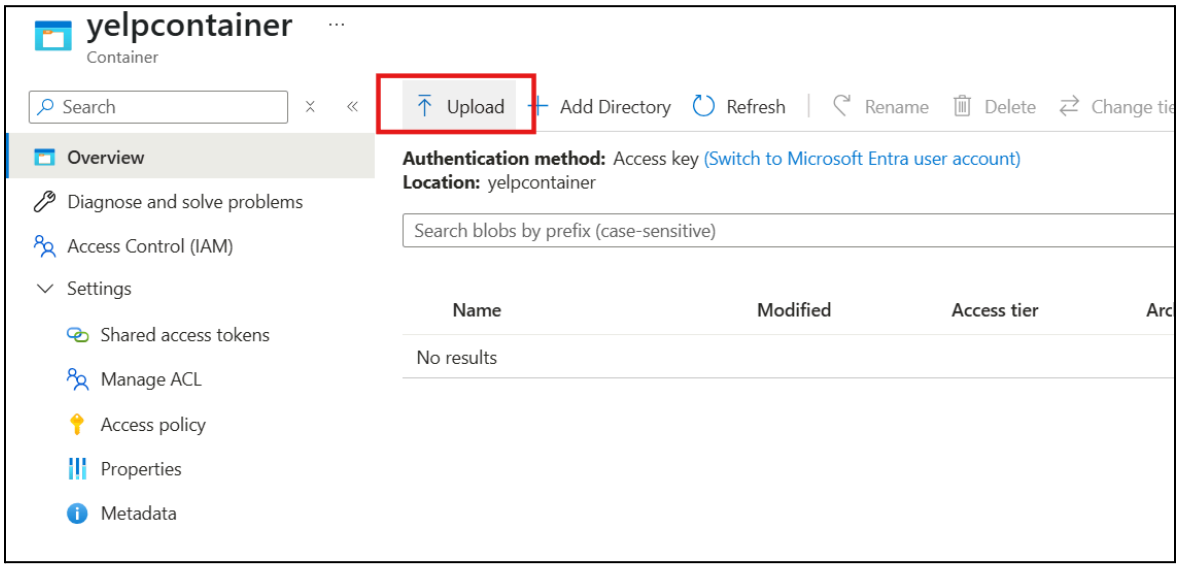
This will disable soft delete for both blobs and containers in the ADLS Storage Account.

The reason behind disabling this option is to get more control over when we want to terminate the present in the Storage Account and container.

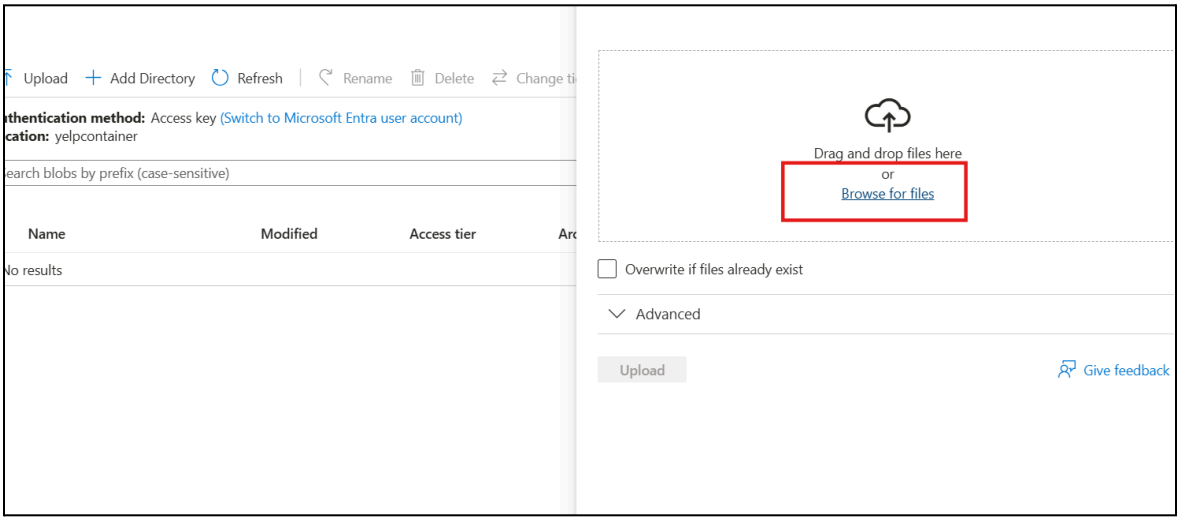


Uploading Files: (yelpcontainer)

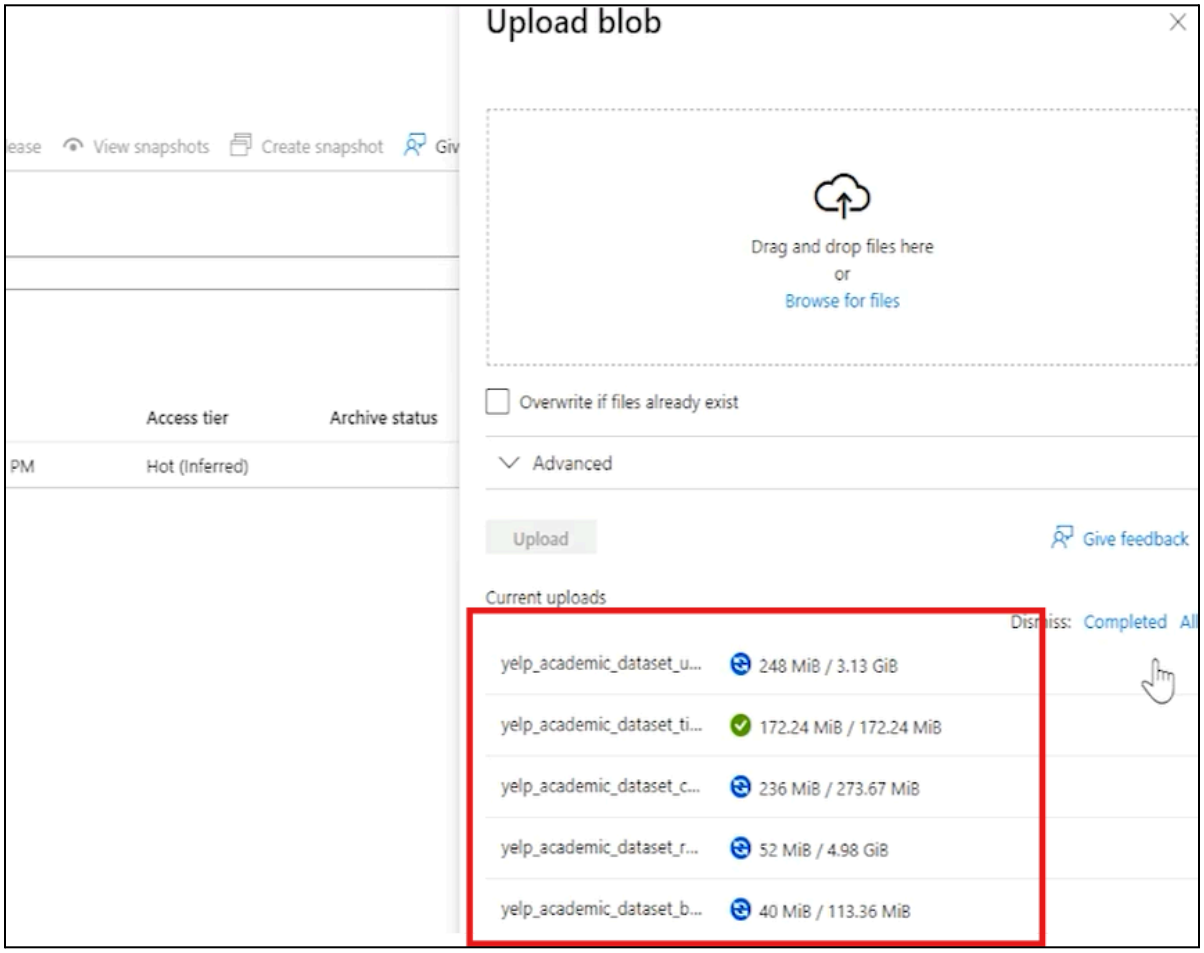
- Download the dataset from the data folders of this project - Dataset 1 and Dataset 2
- Unzip the downloaded files on your local system.
- After unzipping, go to the **first container** (yelpcontainer) that you previously created to store the raw Yelp JSON files.



- Click on **Upload**, then **browse for files** to locate the unzipped files on your local system.

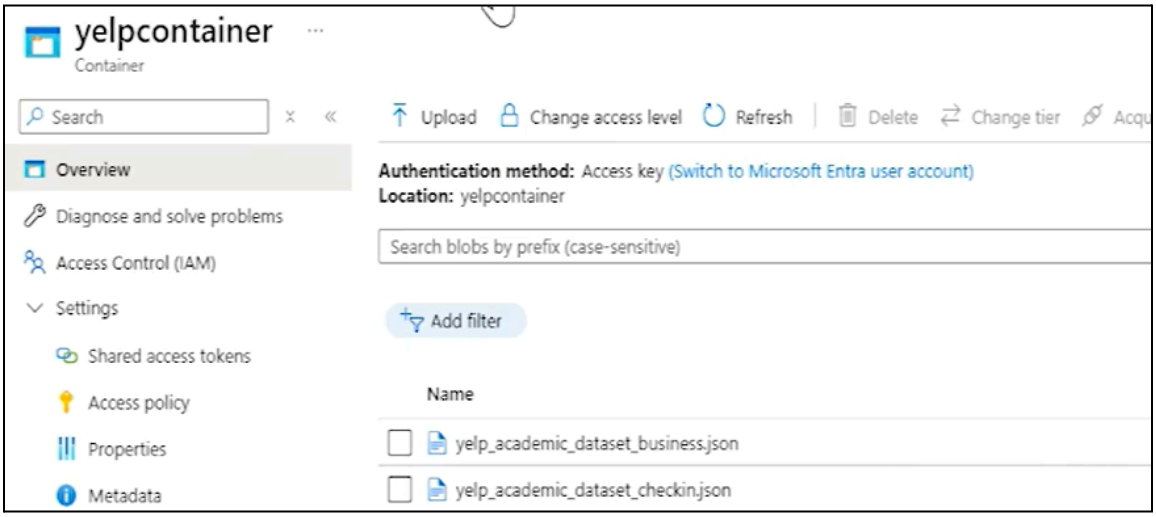


- Select all the JSON files from both data folders so that every Yelp JSON data file is prepared for upload.



- Once the files are selected, click **Upload** to begin the transfer of all selected files to the first container (yelpcontainer).

- All the Yelp JSON files will now be uploaded seamlessly to the first container.

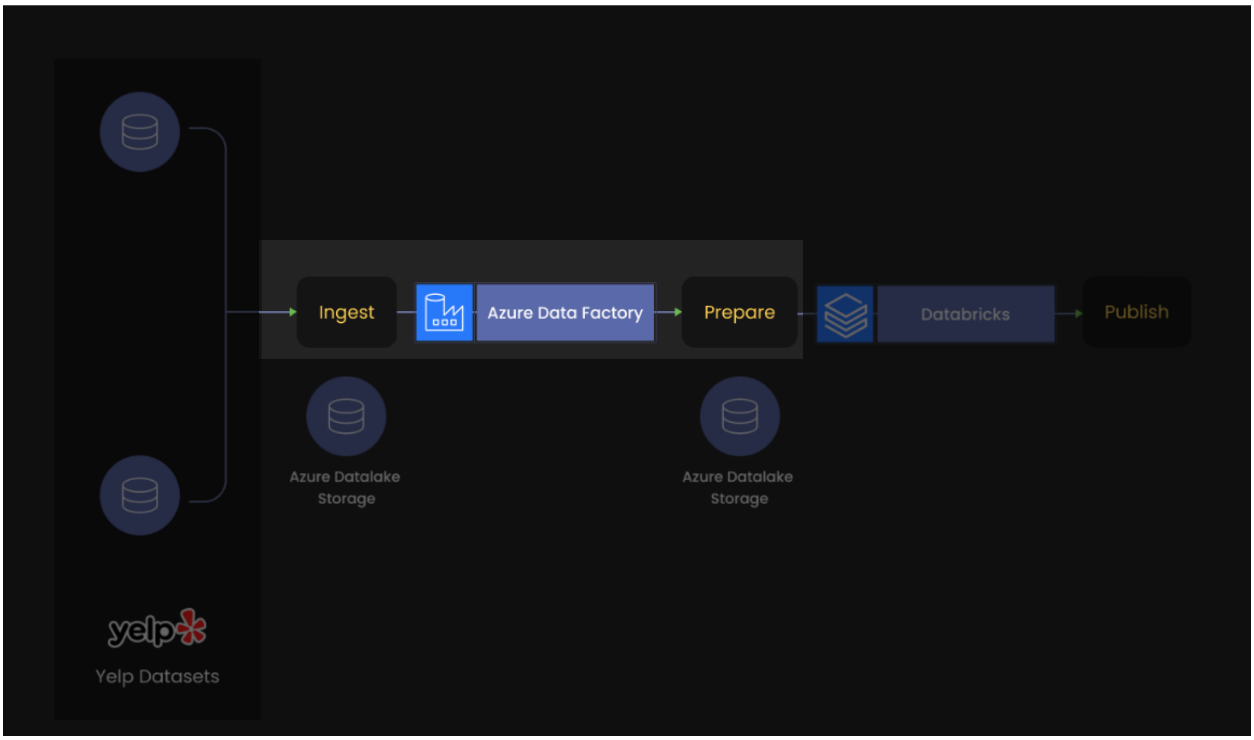


Later, when we create the **copy pipeline** in Azure Data Factory (ADF), this pipeline will copy all the files from the first container **(yelpcontainer)** and transfer them to the **second container (sourcecontainer).**

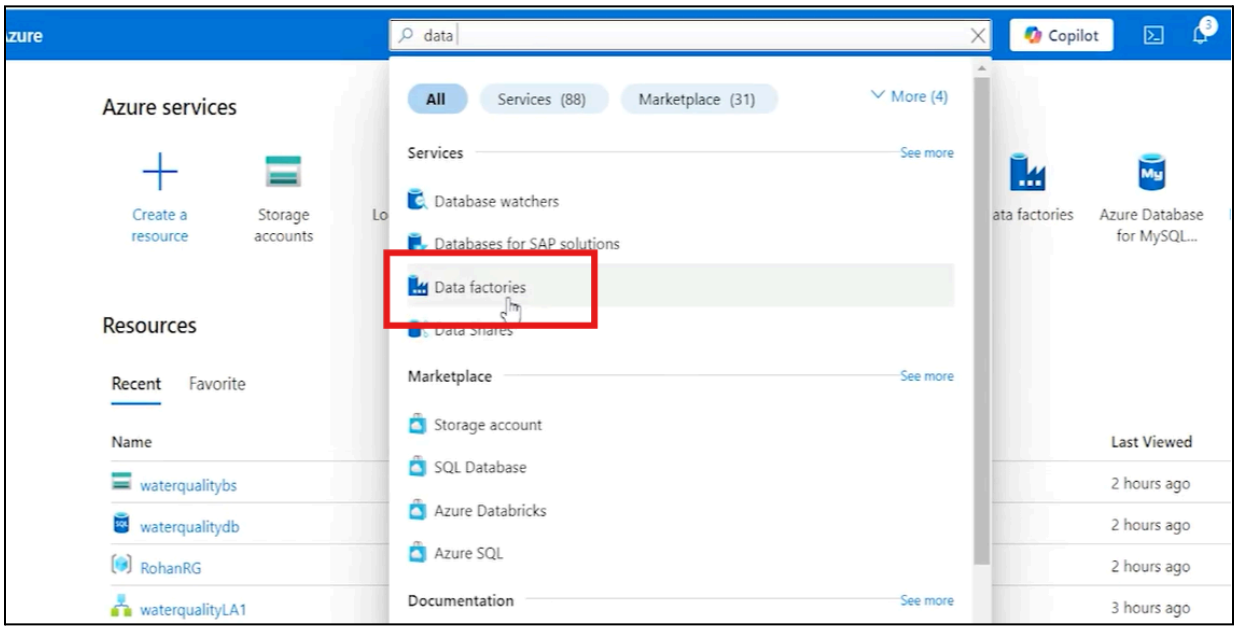
From the second container, a **Databricks Spark job** will be created to read these files and load them into a DataFrame for further processing.

Creating Azure Data Factory

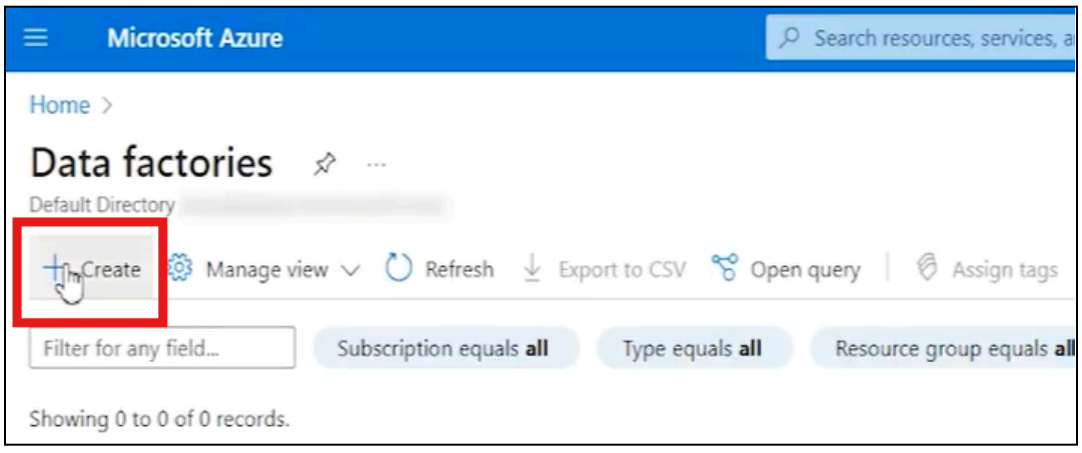
In this project, a copy pipeline will be created within Azure Data Factory (ADF) to transfer data from one container of ADLS Gen2 to another ADLS container. This pipeline will ensure that a JSON data file is copied to the ADLS container.



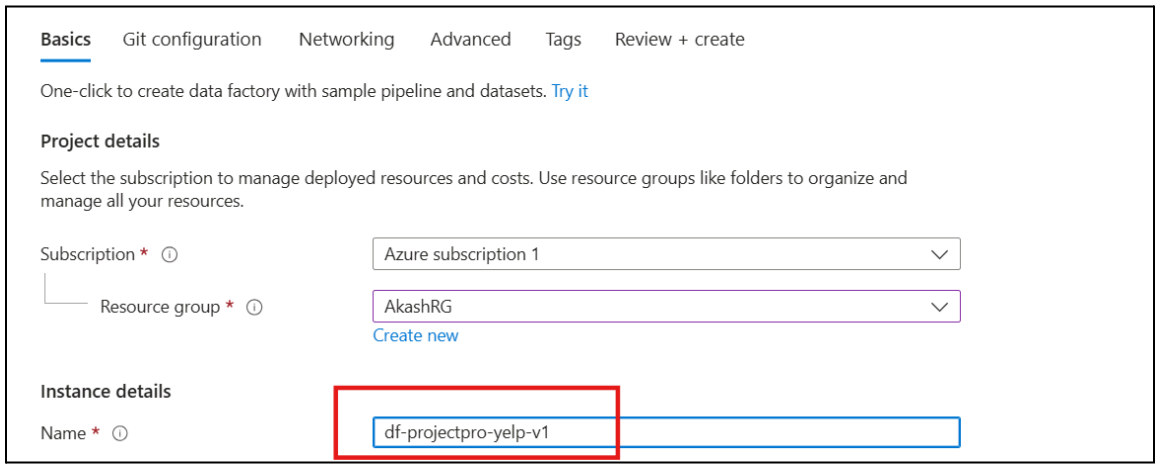
a) Navigate back to the Azure portal and in the search bar, type **Azure Data Factory**. Select **Data Factories** from the suggestions.



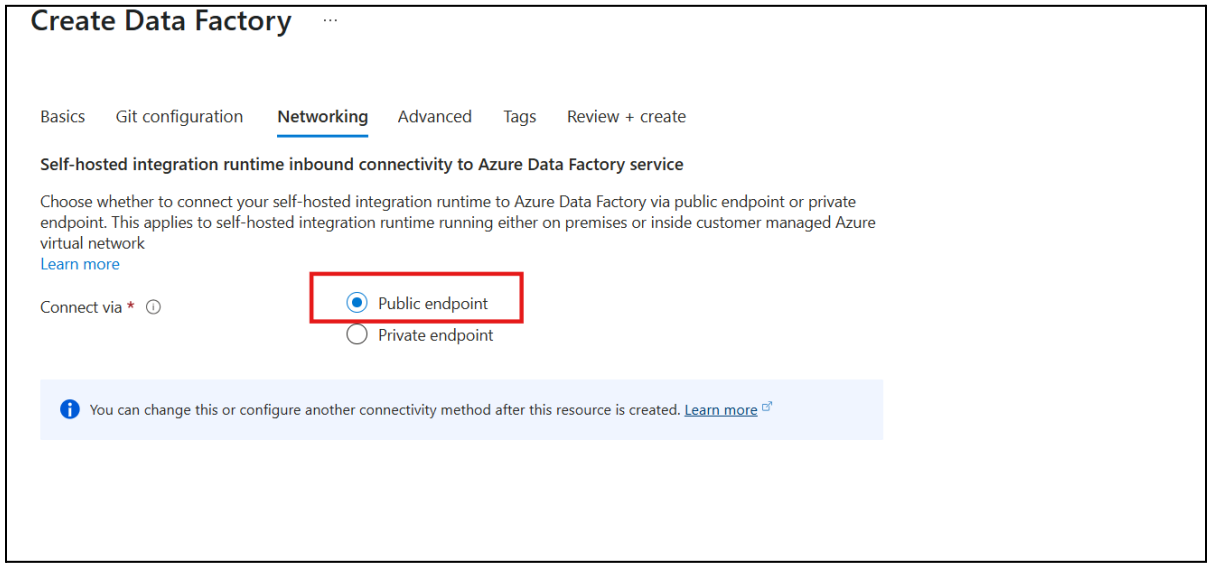
b) Click on **+Create**



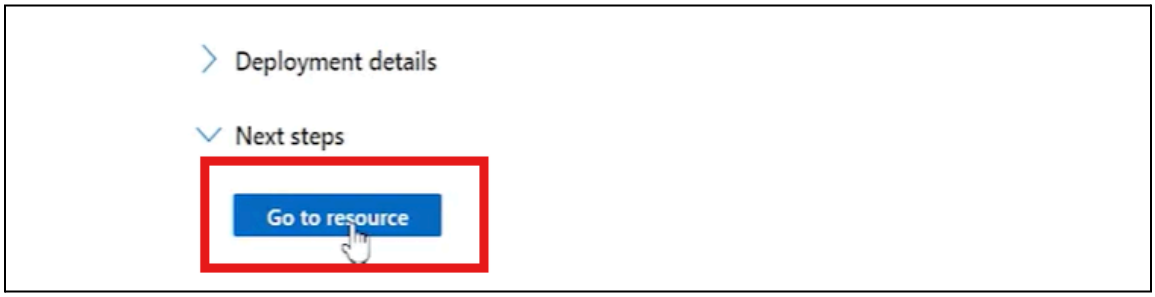
c) In the **Resource Group** field, select the resource group you created from the dropdown. Also, specify the name of the Azure Data Factory as you want.



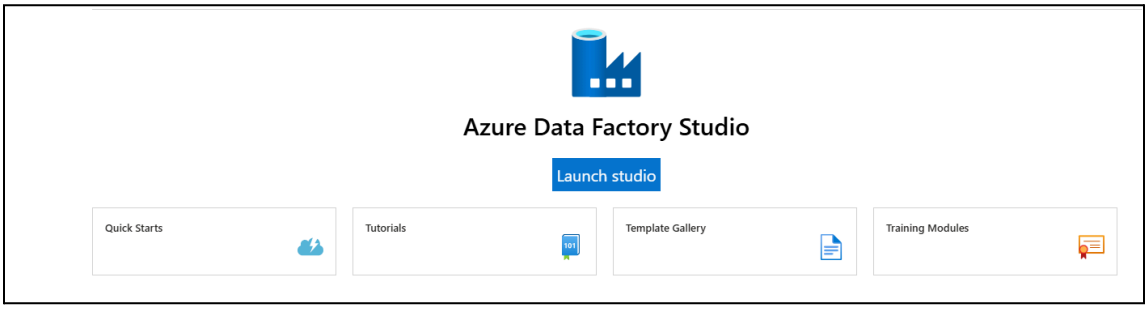
d) Then click on **Next, and in the Networking tab,** make sure the public endpoint is selected.



e) Click **Next** two times, then click **Review + Create**. The validation of deployment might take some time. Once the deployment is successful, click **Go to Resource**.



f) Click on **Launch Studio** to open the Azure Data Factory interface.



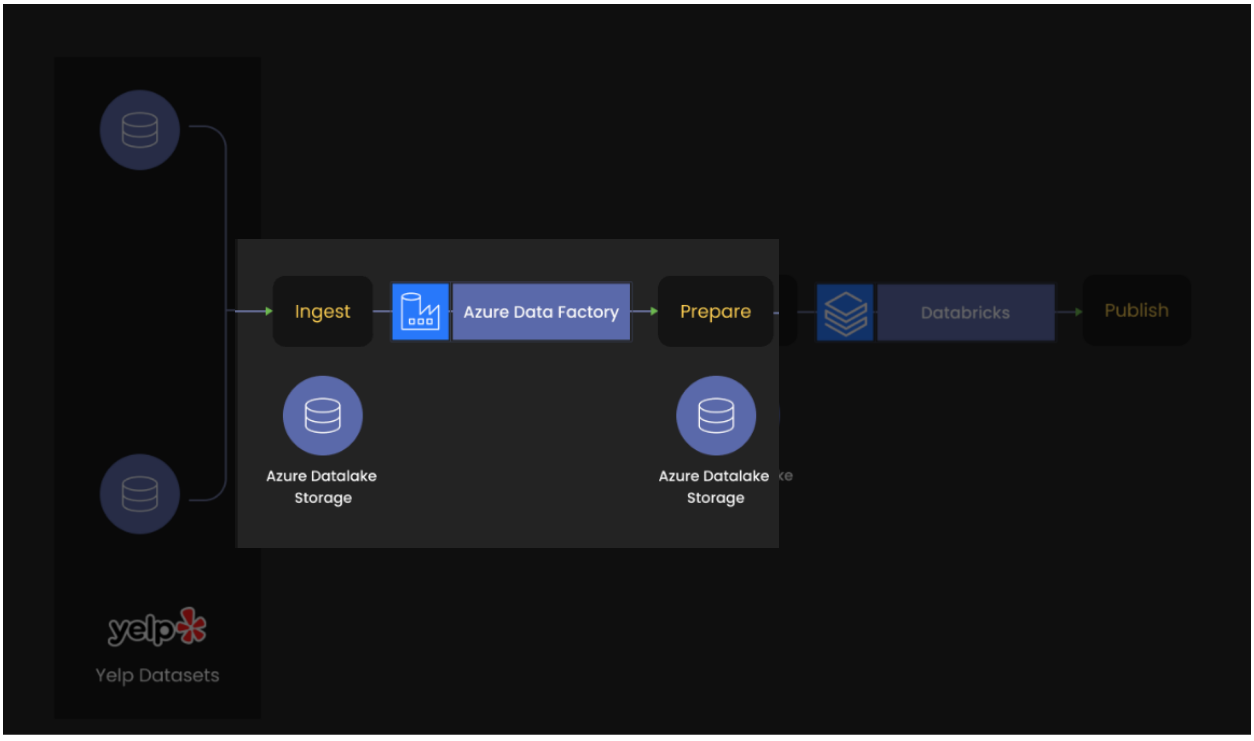
Copy Pipeline

The next step is to create a **copy pipeline** where different configurations and components will be set up to enable the data transfer. This pipeline will be responsible for copying the data from the **first container** (yelpcontainer) to the **second container** (sourcecontainer).

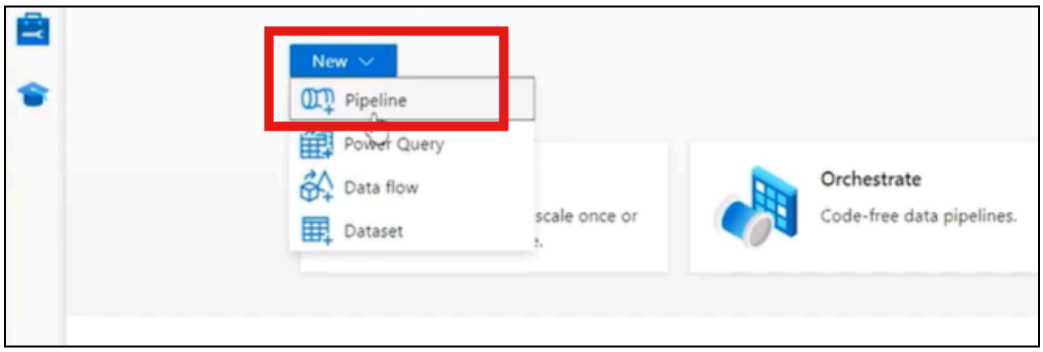
- The pipeline will use two **Linked Services**: one for connecting to the source container and another for the destination container, both within the same ADLS Gen2 storage account.

- A **Copy Data activity** will be added to the pipeline, which will define the logic of reading data from the first container and writing it to the second.

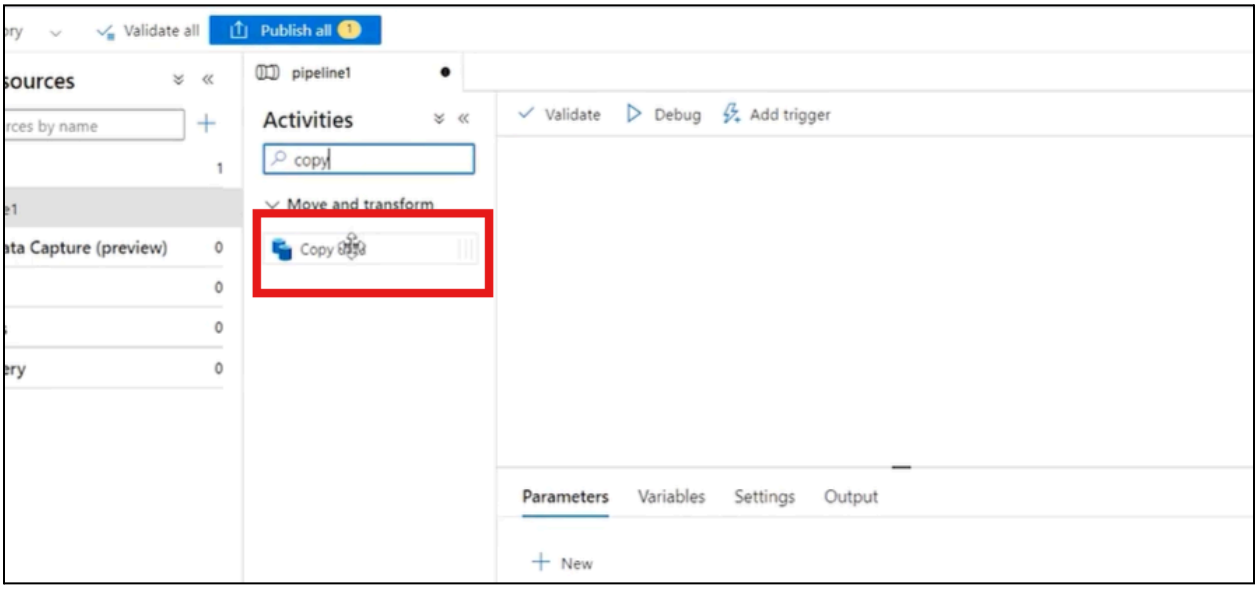
This configuration ensures an automated and efficient movement of data from the original source to a staging location that Databricks will later use.



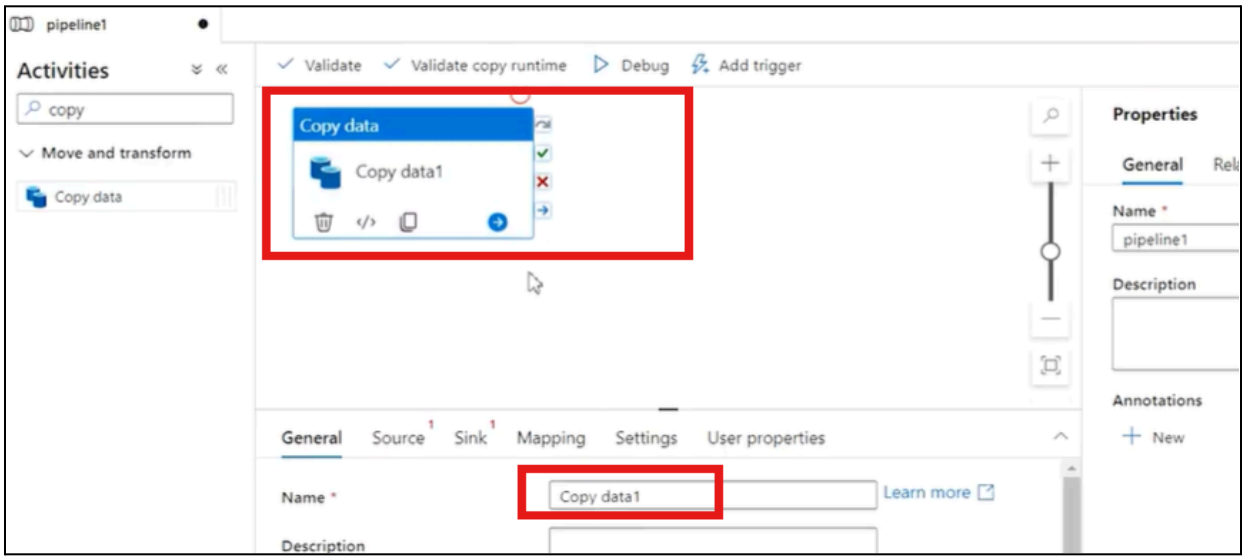
a) Click on **New** and then click the **+Pipeline** option.



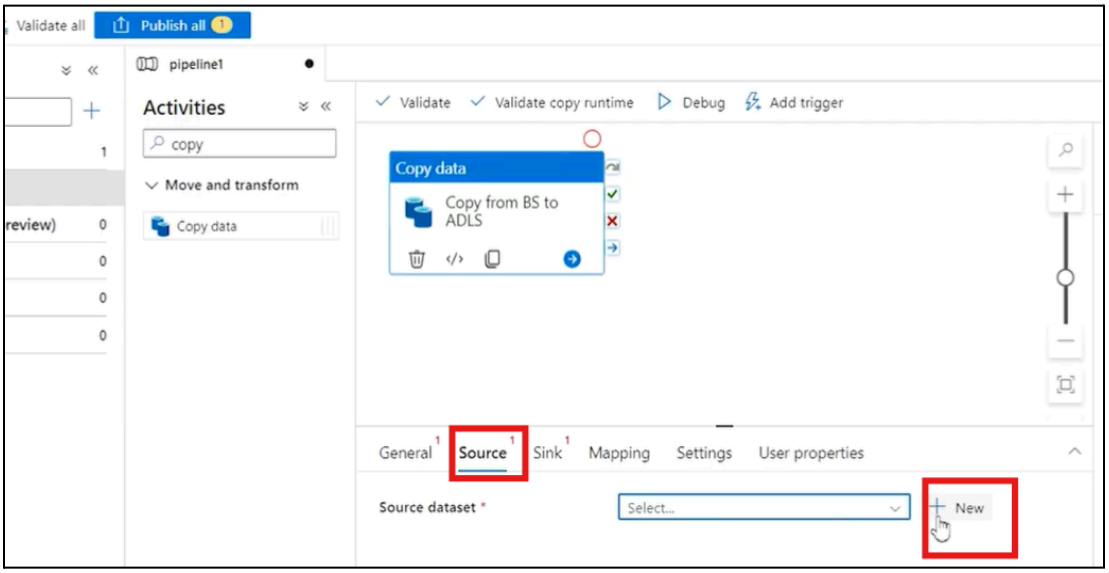
b) In the **Activities** section, click on the **Move & Transform** option, then click on the **Copy Data** pipeline and drag it to the white canvas on the right side.



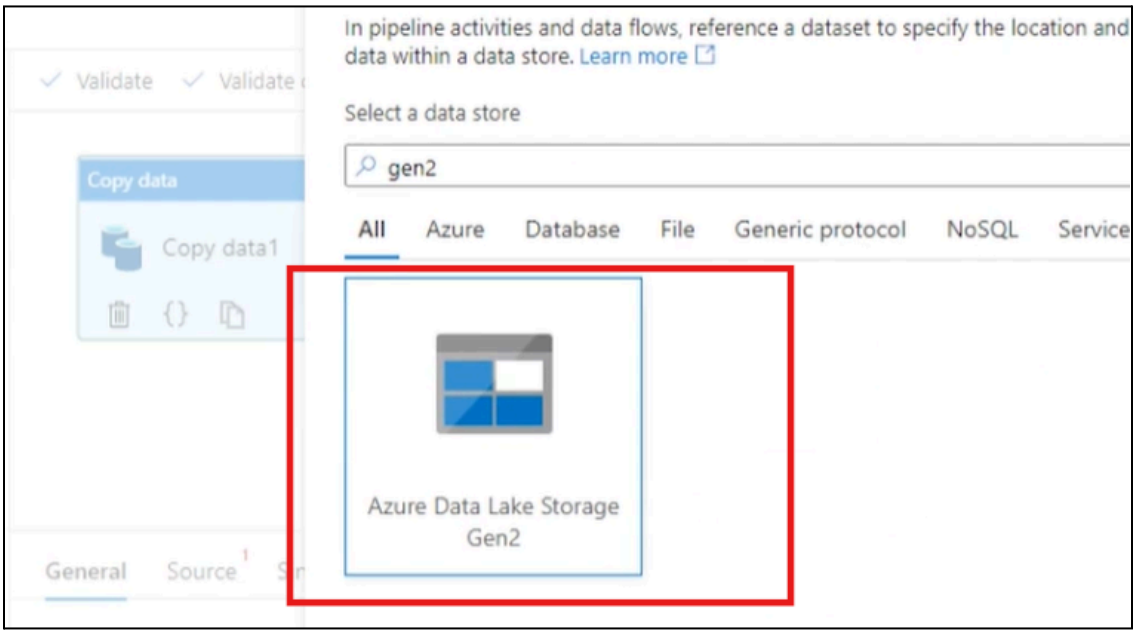
c) Click on the **Copy Pipeline** on the white canvas, then in the **General** section, specify the name of the pipeline as desired.



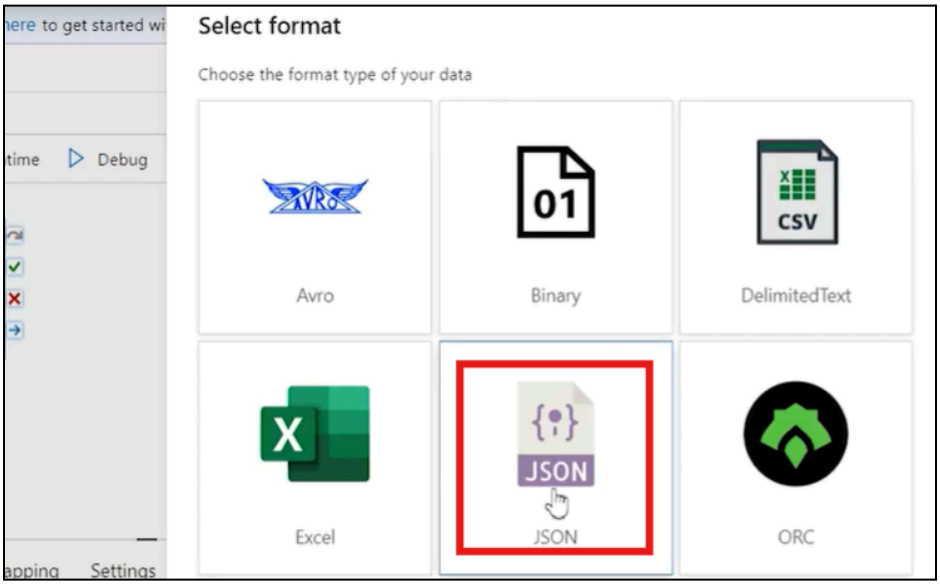
d) Click on the **Source** section, then click on **New** to set the new source. As discussed, the source is the **ADLS gen2 Container (yelpcontainer)** containing the JSON data file.



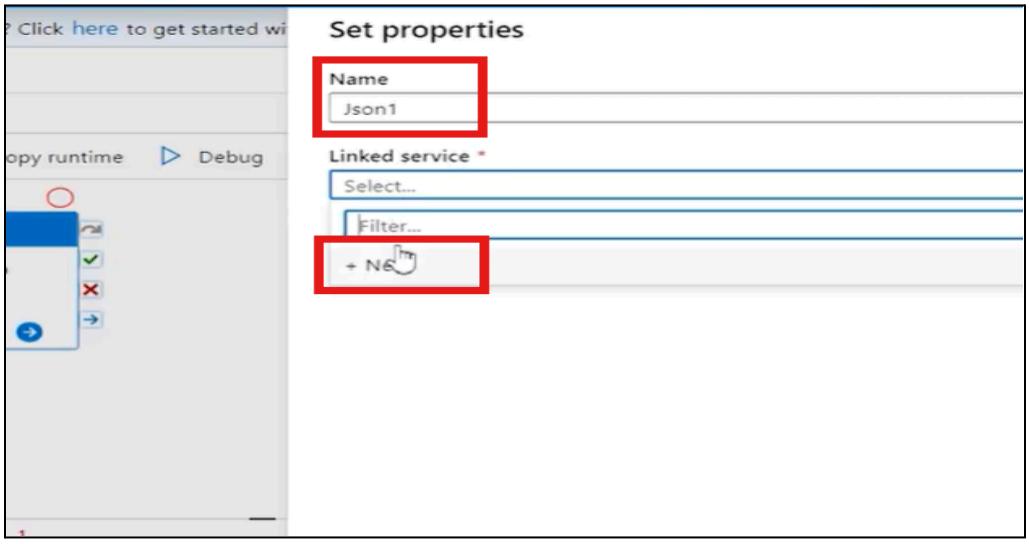
e)In the right-side panel, under the **Select a data source** field, type **Azure Data Lake** and select the **Azure Data Lake Storage Gen2** option. Then, click on **continue**



f) From the select format option, click on **JSON Format** and click on **Continue.** Since the data file in the container is in JSON format.

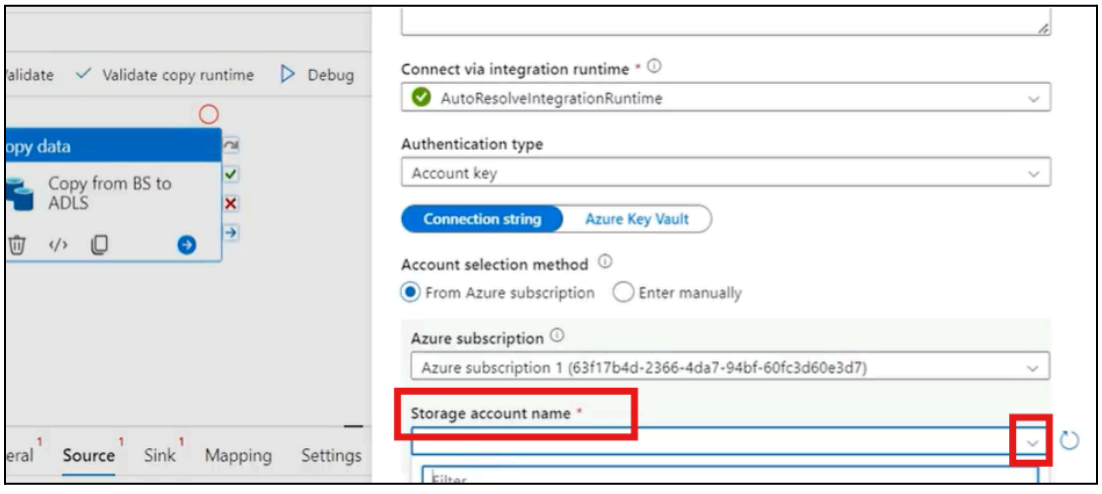


g) Give a name to the source and then click on **+New** to create a linked service for the Azure Data Lake Storage container.



h) In the **New Linked Service** window, give any name to the linked service and keep the default settings, and then, from the **Storage Accounts** dropdown, select the Azure Data Lake Storage account you created.

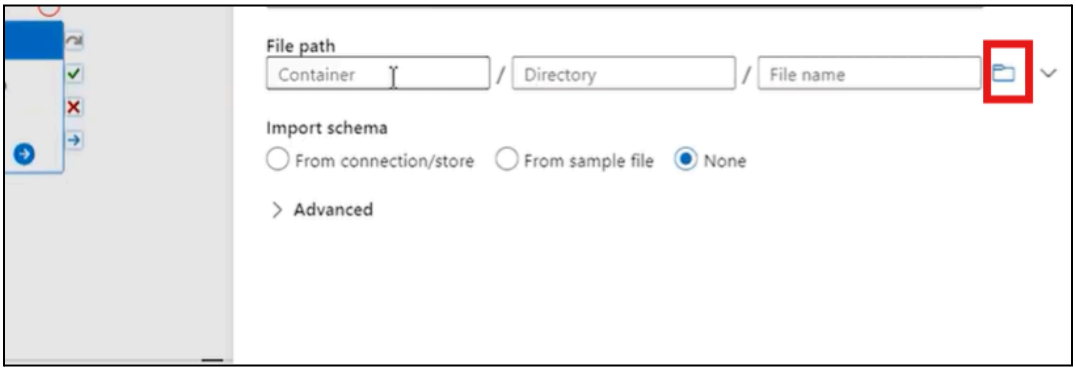
Then click on **"Create"**



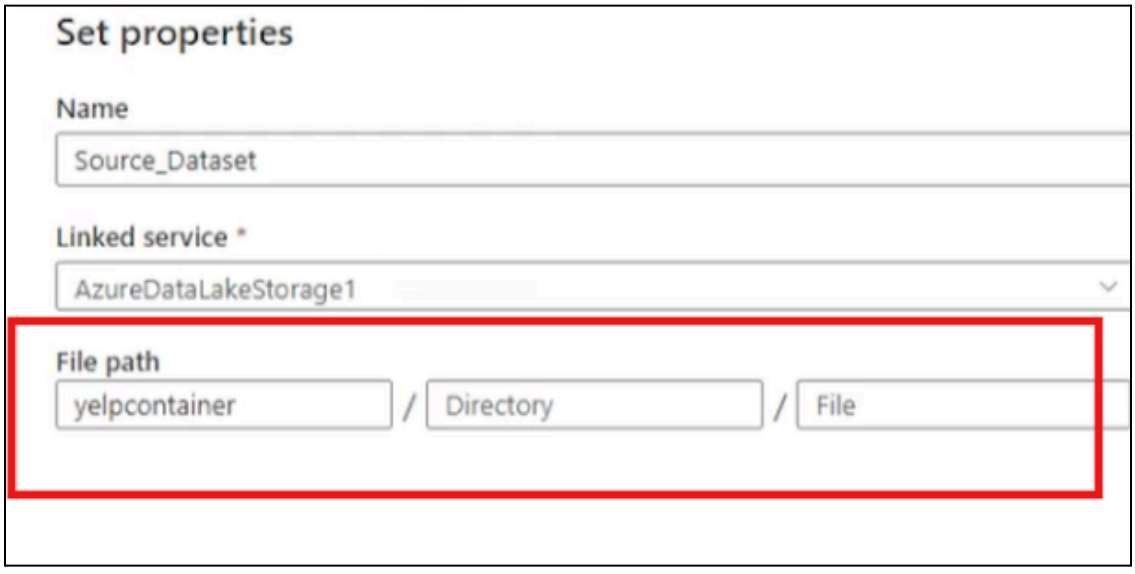
This will create a linked Service between Azure Data Lake storage Gen2 and ADF.

i) Now, **set the properties** to define the exact location where the JSON file is located in the Azure Data Lake Storage container.

Click on the **file icon** to browse the location of the Azure Blob container that has **a JSON file.**

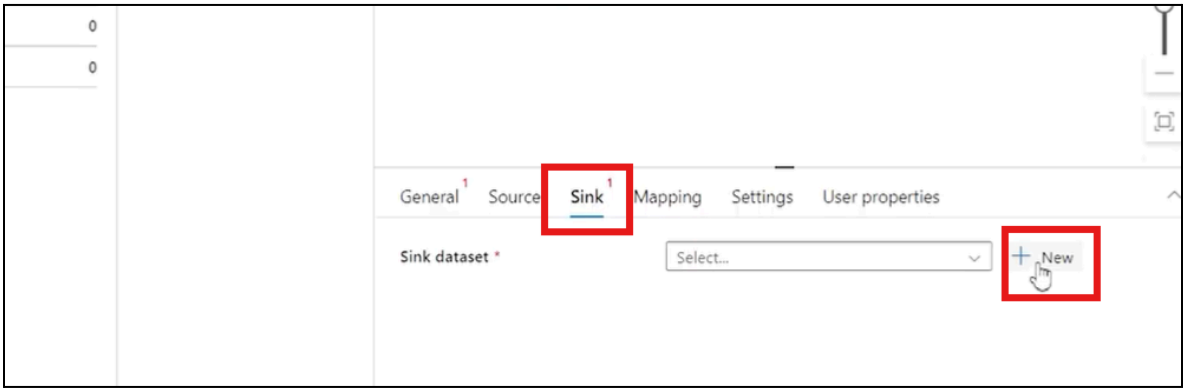


j) Then click on the **Azure Data Lake Storage container (yelpcontainer)** you created, and then click on **OK.**

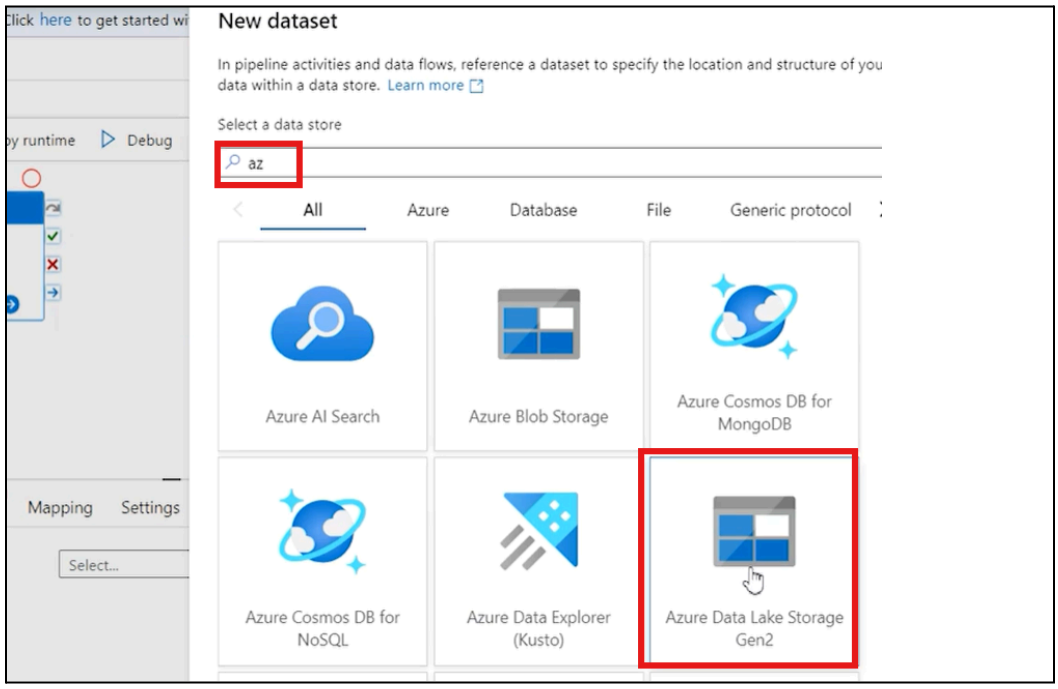


The reason for specifying the container only, rather than a particular JSON file, is that the main objective of this pipeline is to transfer any JSON file that lands in the Azure Data Lake Storage container to another Azure Data Lake Storage (ADLS) container. This approach ensures that the pipeline can handle multiple files dynamically and seamlessly, rather than being restricted to a specific file

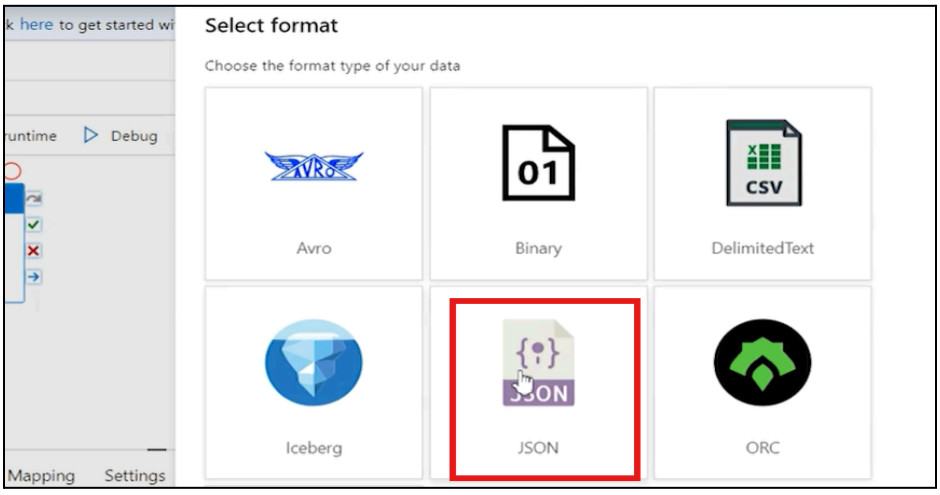
k) Click on Sink to set up the **ADLS Gen2 container** as a sink to store real-time JSON files that are currently in the Azure Blob Storage Container.



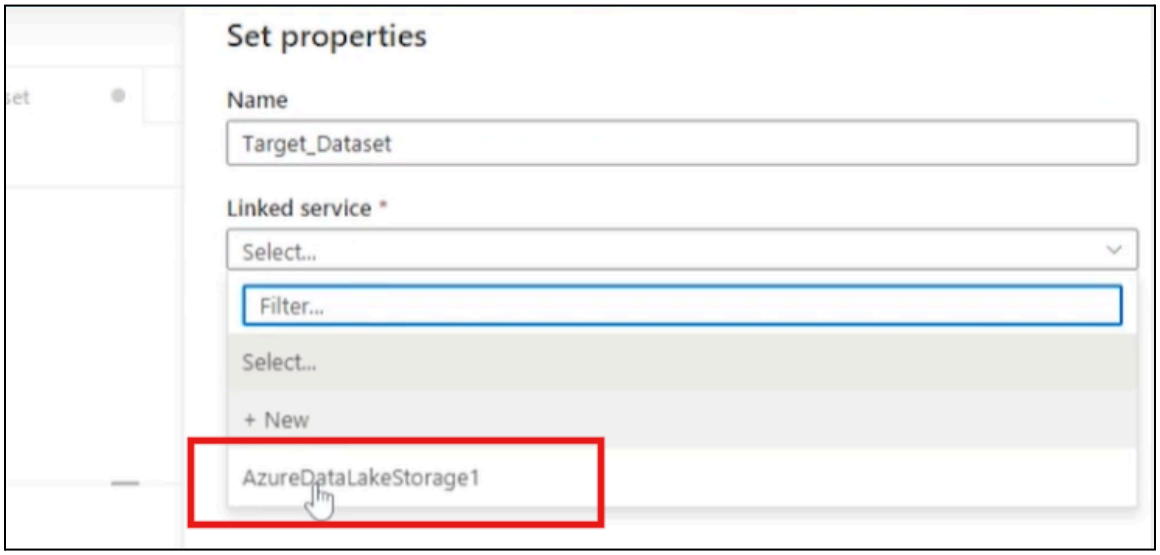
l) In the right-side panel, under the **Select a data source** field, type **Azure Data Lake Storage Gen2** select the **ADLS Gen2** option, and click on **continue.**



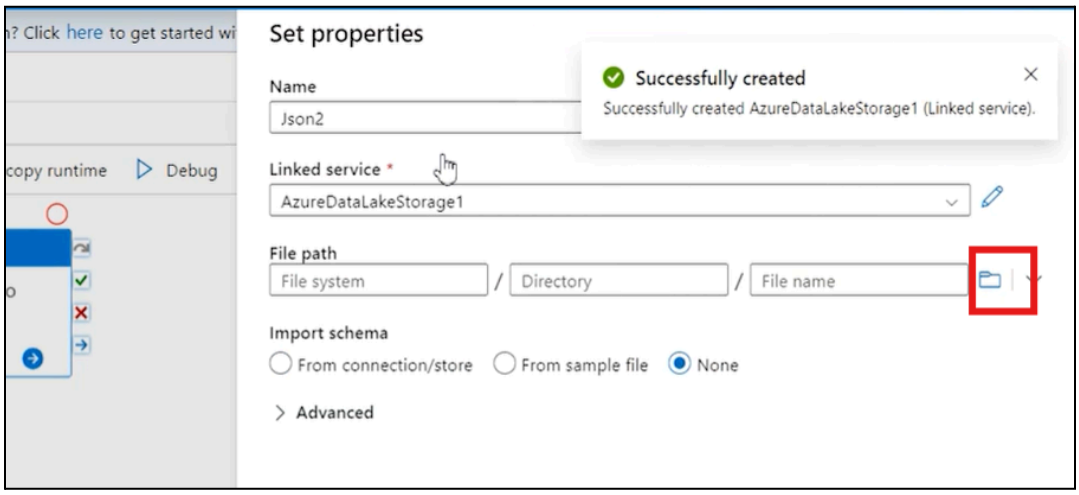
m) From the select format, select **JSON** as a new format, and then click on **Continue.**



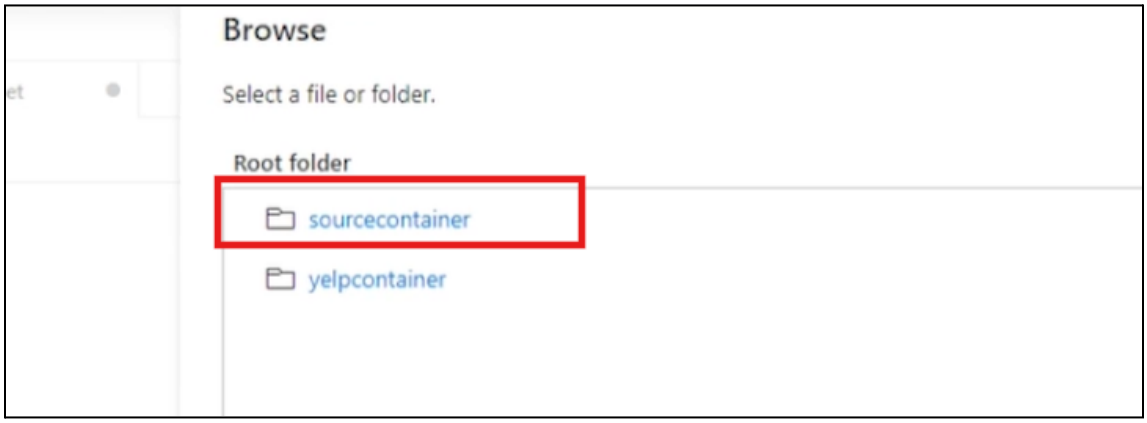
n) In the Linked Service configuration, select the same Linked Service that was created earlier, as this project uses the same Azure Data Lake Storage account throughout.

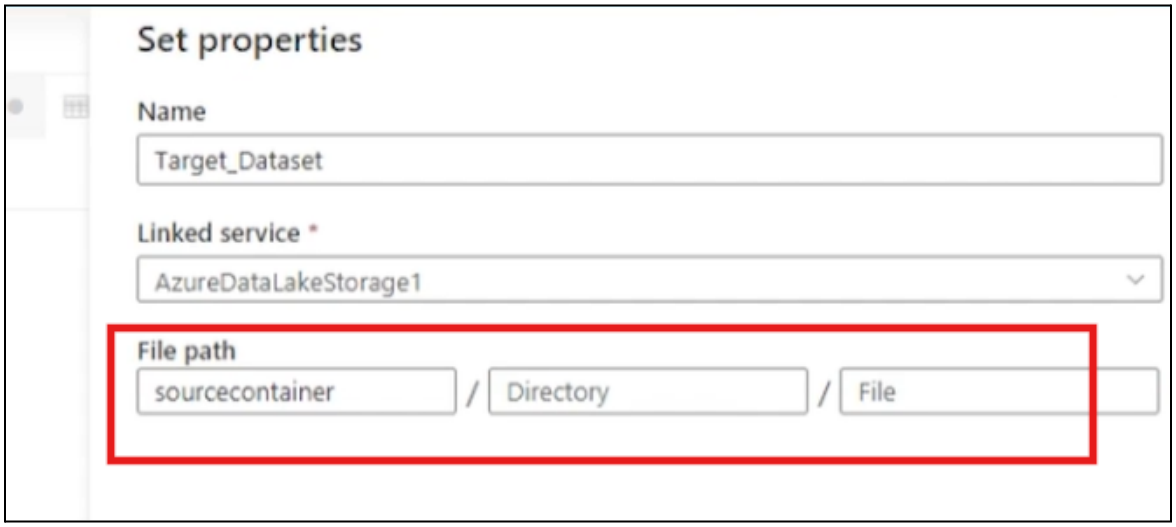


o) Then, from this panel, specify the details, click on the file icon, and browse to the destination ADLS container **(sourcecontainer)** that you created earlier.

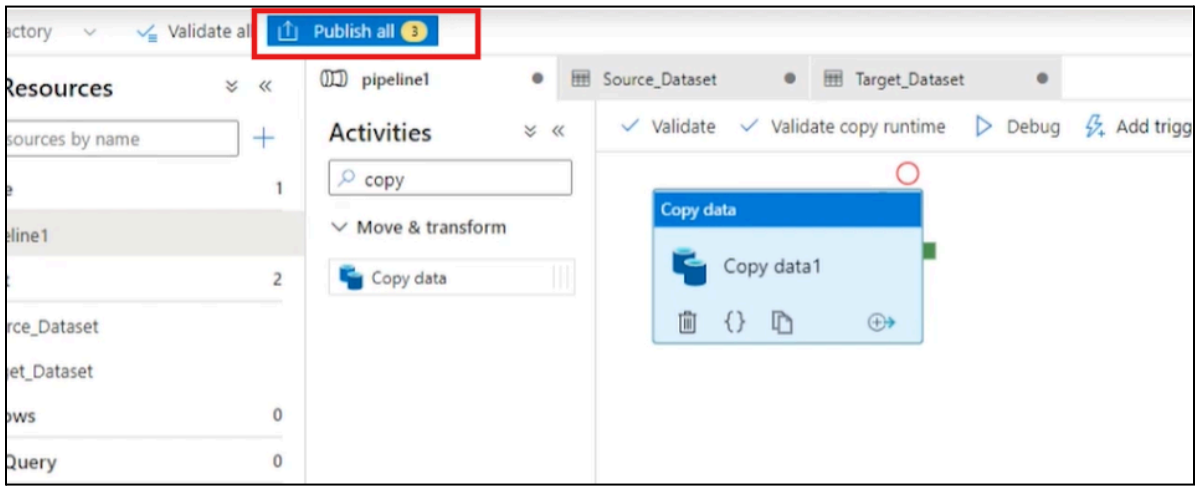


p) Select the **ADLS Comatiner** that you created and then click on **ok.**



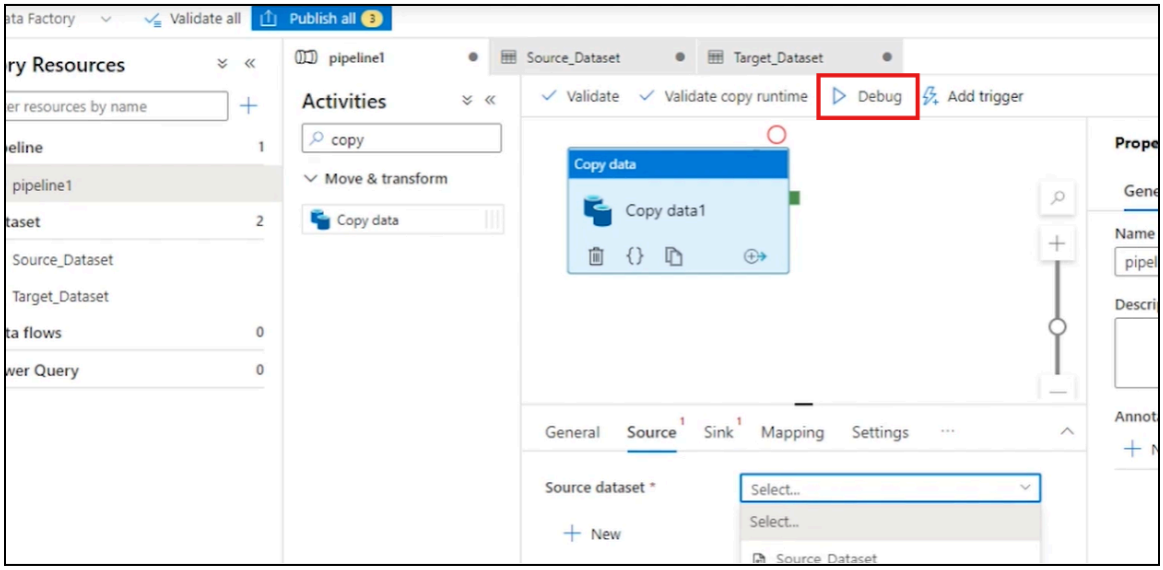


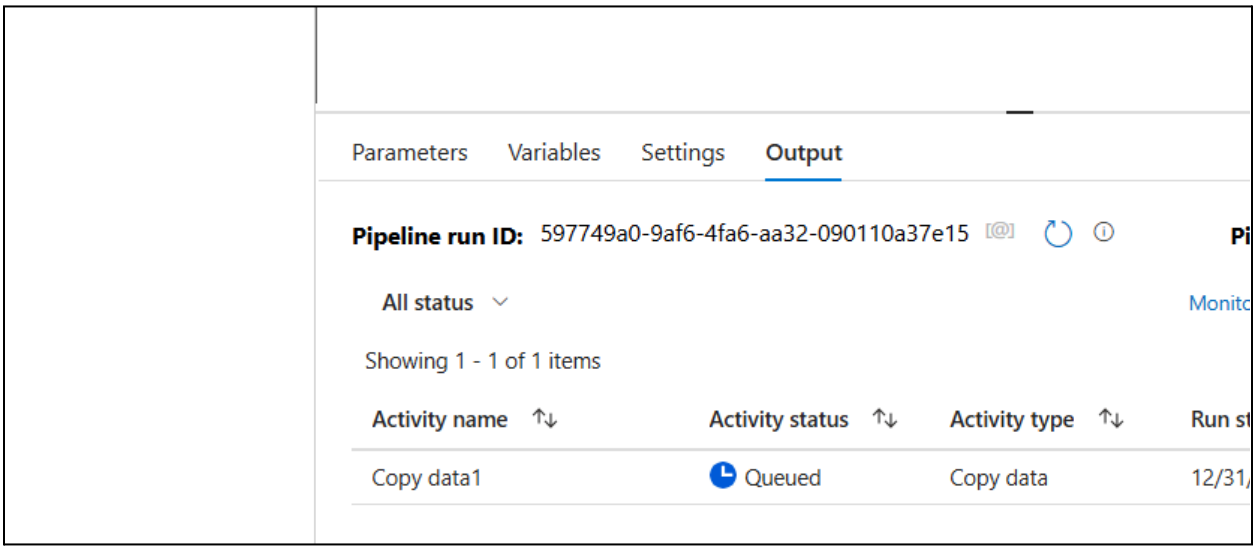
q) Then, click on the **Publish** option. Publishing in ADF ensures that all pipeline configurations are saved correctly and that all connections are provisioned, making the copy pipeline ready for execution.



After that, click on the **debug** option, and it will trigger the **"Copy pipeline"**.

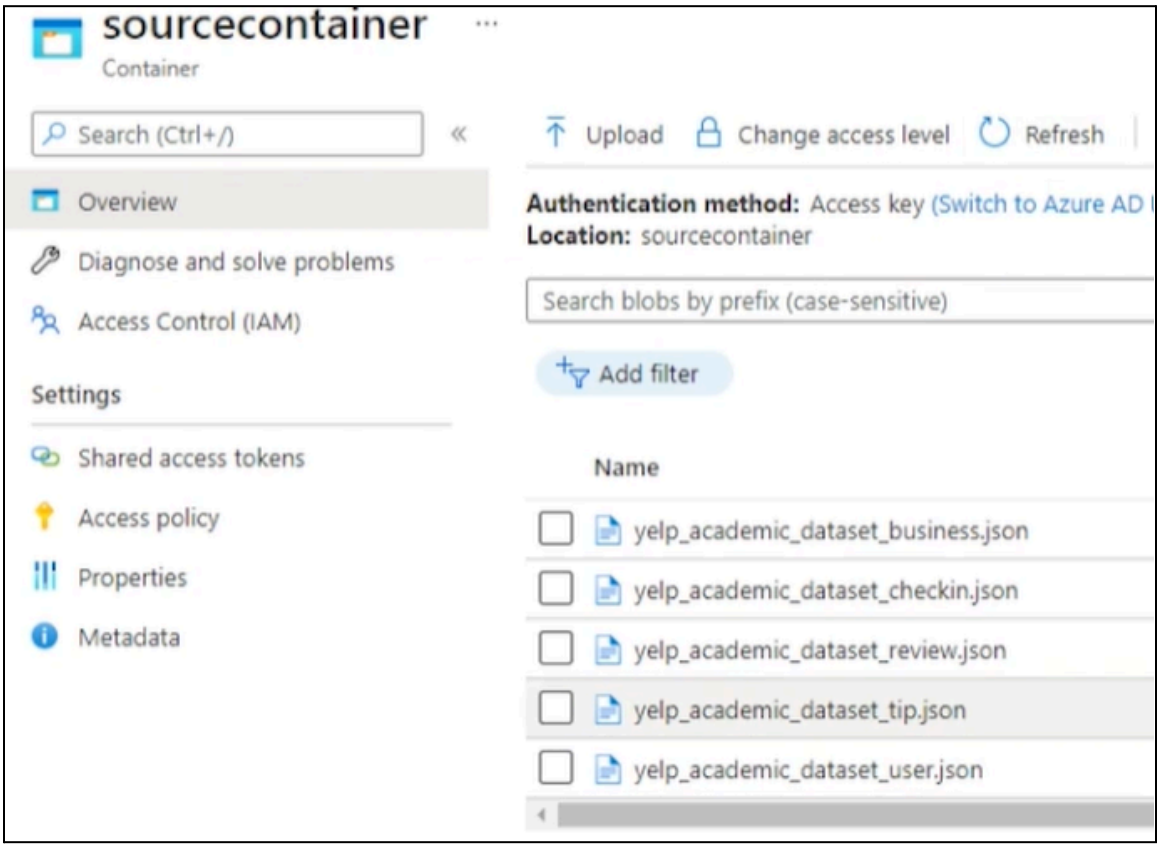
As a result, the pipeline will be executed successfully.





After the successful execution of the copy pipeline in Azure Data Factory (ADF), which updates the monitoring status from *Queued* to *Succeeded*, the data files are copied from the source container to the sink container.

- To verify this, navigate to the **sourcecontainer**, where you will now see all the Yelp JSON data files that were copied from the **yelpcontainer** (which initially served as the source container).



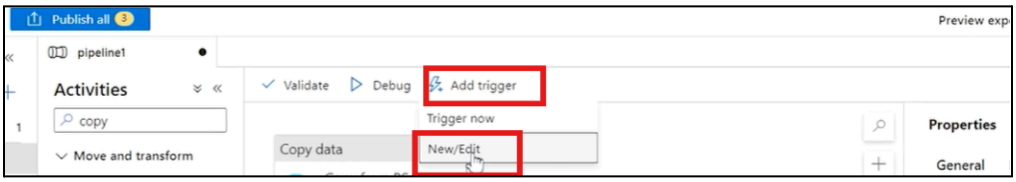
- This confirms that the copy pipeline has completed successfully, and the data is now ready to be accessed by downstream processes like Databricks

Automation in Copy Pipeline

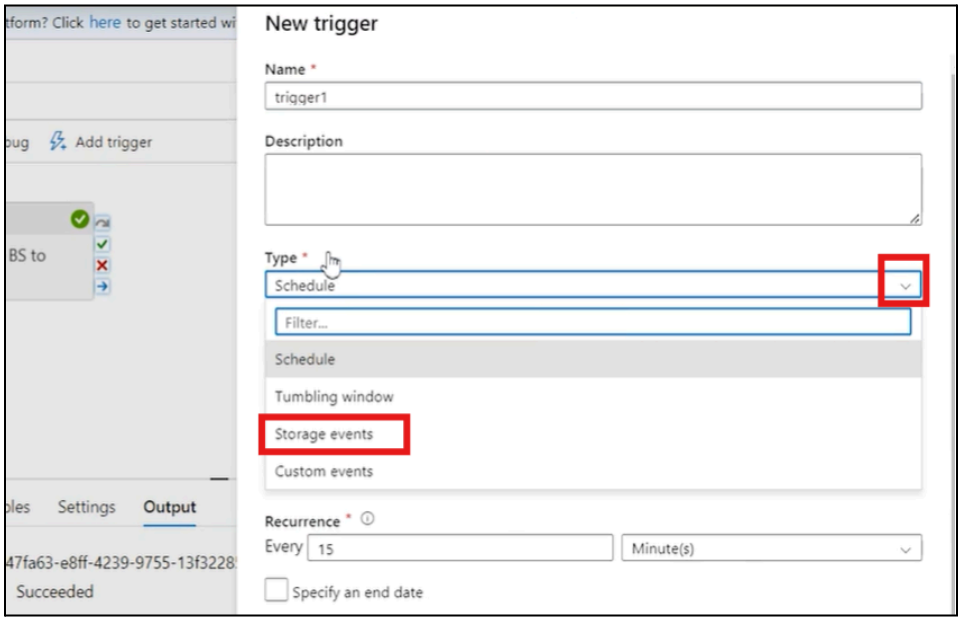
To ensure the automation of the real-time pipeline in Azure Data Factory (ADF), it is essential to eliminate manual intervention and add automation to the copy pipeline.

Since real-time pipelines deal with continuous data ingestion, where JSON files are consistently loaded into the Azure Data Lake Storage Container, these files must be automatically moved to another Azure Data Lake Storage (ADLS) container **(sourcecontainer)** as soon as they are created in the source **Azure Data Storage Container (yelpcontainer).** To achieve this, a trigger should be implemented in the ADF copy pipeline.

r) In the top panel of Azure Data Factory, click on **"Trigger"** and then select **"New/Edit"**. From the dropdown, choose **"New Trigger"** to create a new trigger for your copy pipeline.



s) In the **Type** field, select **"Storage Events"** from the dropdown as the trigger type.



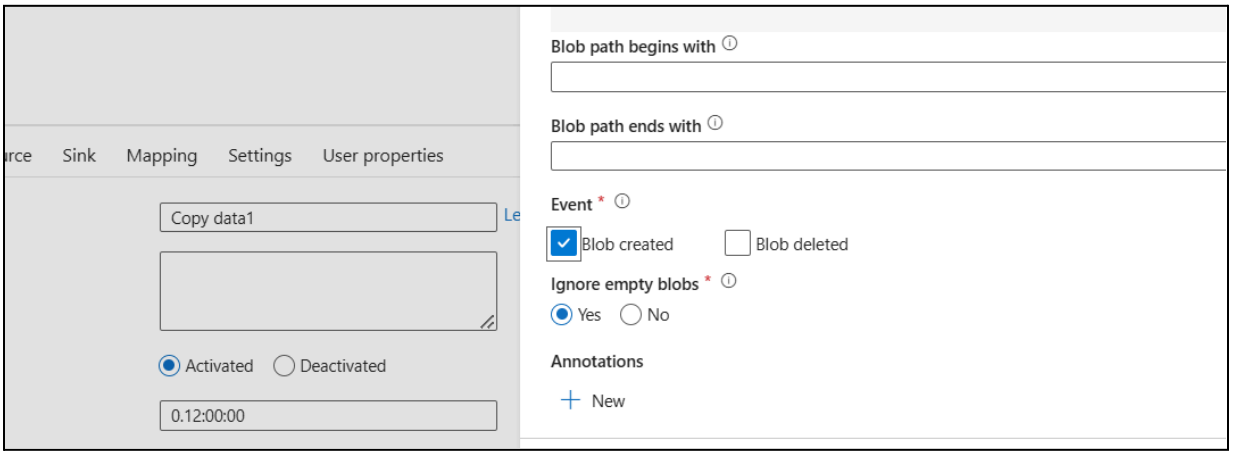
The reason for choosing **Storage Events** as the trigger is that it will automatically activate whenever an activity occurs in the source **Azure Data Lake Storage container (yelpcontainer)**, such as when a new data file lands in the container. This ensures that the pipeline is triggered in real time

t) In the **Storage account** field, select the Azure Data Lake Storage account you created from the dropdown.

Next, in the **Container name** field, choose the Azure Data Storage container you created (**yelpcontainer**) from the dropdown.

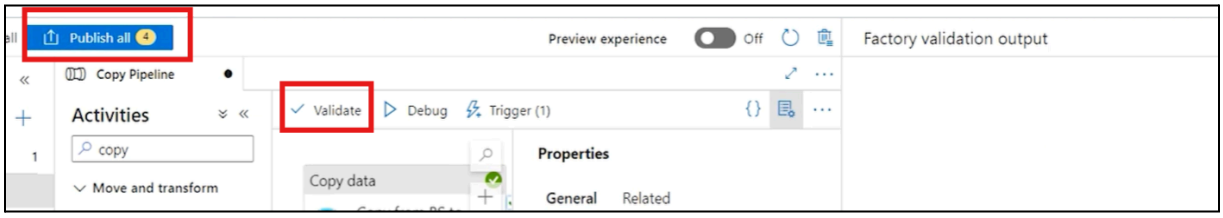
This is the container where JSON data files are received.

u) In the **Event** section, select **Blob Created** and then click on **Continue two times, and then click on "Ok".**



Selecting Blob Created specifies that the copy pipeline will be triggered automatically whenever a new data file lands in the Azure Data Lake Storage container. By configuring this, the JSON data file will be seamlessly moved to the destination ADLS container you created in real time, ensuring automation without any manual intervention.

v) Finally, click on the **Validate** and then **Publish** icon to validate all the changes you made to the copy pipeline.



Once published, your copy pipeline will be deployed successfully. Now, whenever a data file lands in the Azure Data Lake Storage container, the pipeline will automatically trigger and load the file into the Azure Data Lake Storage (ADLS) container in real time without requiring any manual intervention.

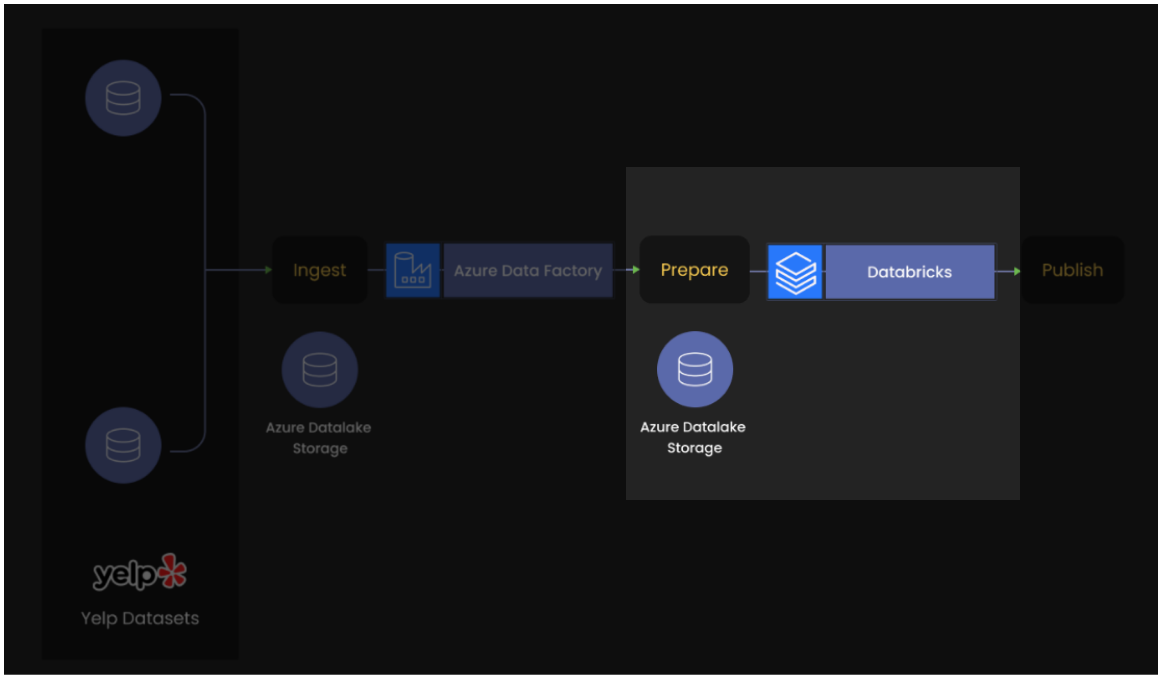
You can test the copy pipeline by uploading a data file to the **Azure Data Lake Storage container (yelpcontainer)**.

Then, you need to navigate to the ADF Interface, and in the left side panel, click on the Monitoring tab. It will showcase the real-time movement of the data file from the source **Azure Data Lake Storage Container (yelpcontainer)** to the **destination ADLS Container (sourcecontainer).**

Creating Azure Databricks

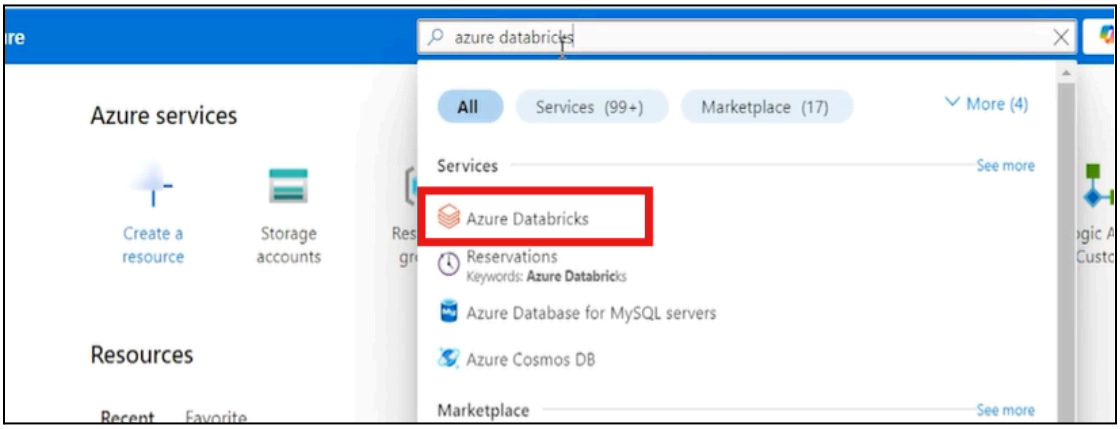
Now, a Databricks workspace and cluster will be created to process the dataset. In this environment, data will be pulled from the second ADLS container **(sourcecontainer)** into Databricks using a Spark job written in PySpark.

- A Databricks workspace and a compute cluster are provisioned to enable data processing.
- The Yelp JSON data, already copied to the second container, is read into Databricks using PySpark DataFrames.
- A comprehensive analysis is then performed on the Yelp dataset, focusing on extracting actionable insights from various business dimensions such as reviews, user behavior, and business attributes.

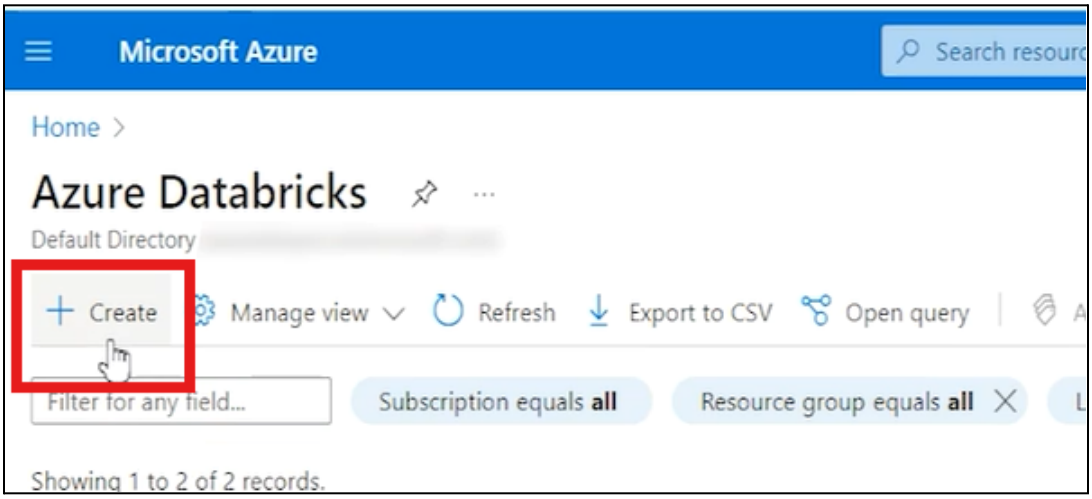


This setup ensures scalable processing and real-time analytics capabilities, leveraging the power of Apache Spark within the Databricks environment.

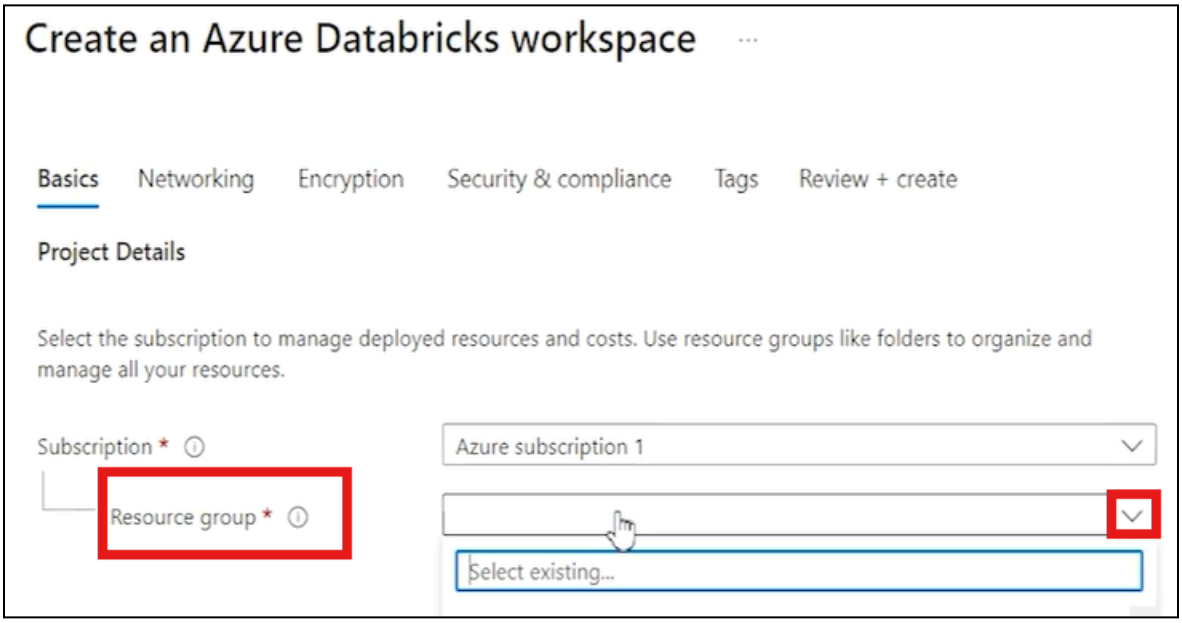
a) Navigate to your Azure Portal and in the search bar, type “Databricks” and select the Databricks service from the suggestions.



b) Click on "**+Create**" to create an Azure Databricks workspace.

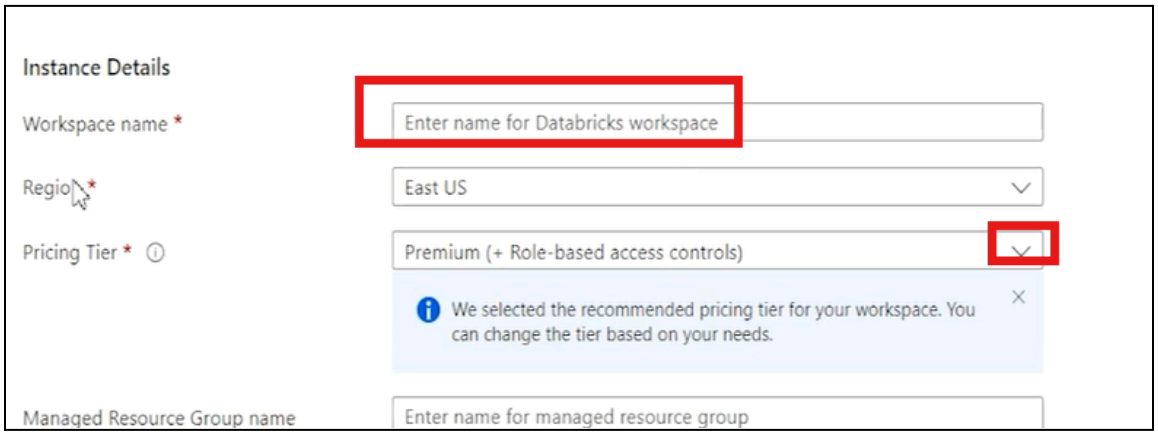


c) In the **"Resource Group"** field, select the resource group you created from the dropdown.

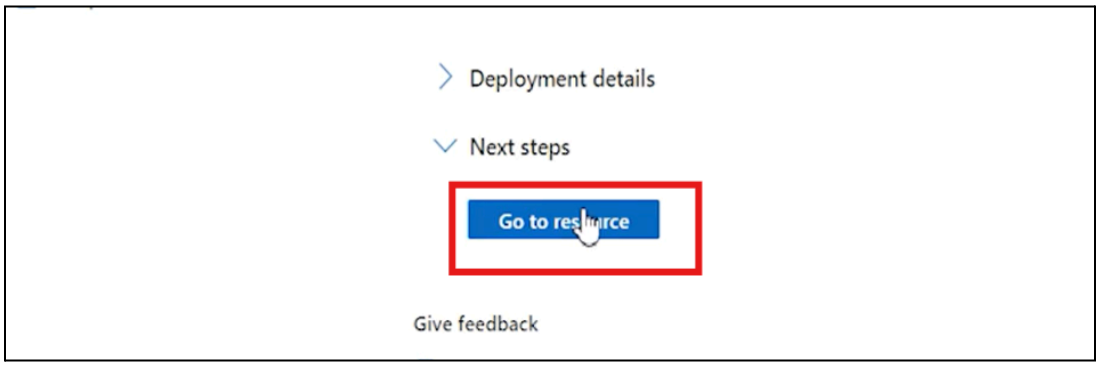


d) In the **"Workspace Name"** field, specify a unique name for your Databricks workspace. Keep all other settings as default. From the pricing tier, select **"standard one"** as premium is costly.

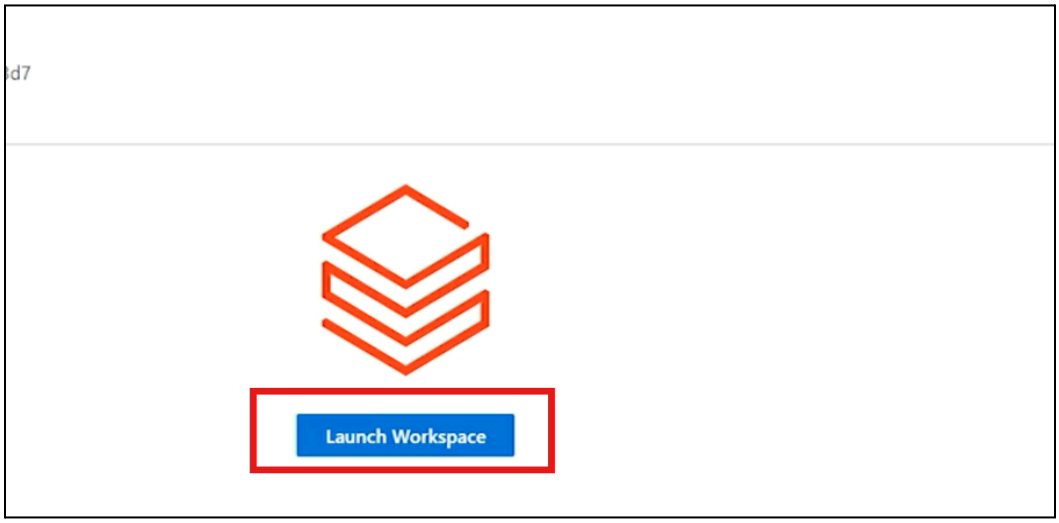
Then, click on **"Review + Create"** and finally click **"Create"** to deploy the workspace.



e) Wait for the deployment of Databricks to complete. Once finished, click on **"Go to resource"** to access your Databricks workspace.

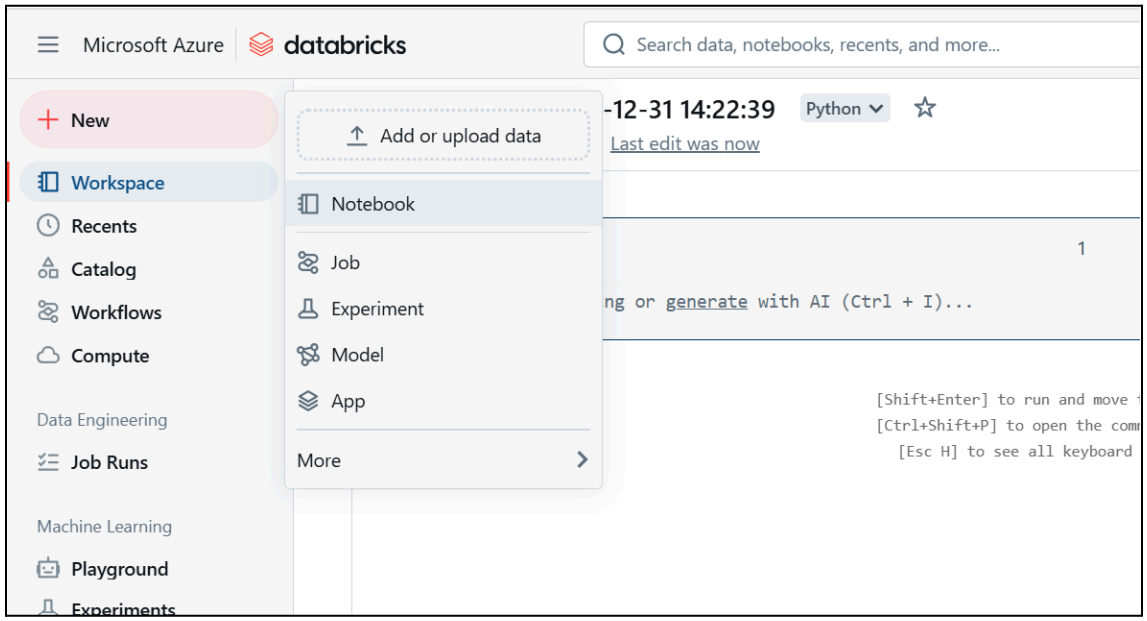


f) Click on the **Launch workspace** to open the Databricks interface. Here, creating a cluster and executing code in a notebook to process the data will be done.

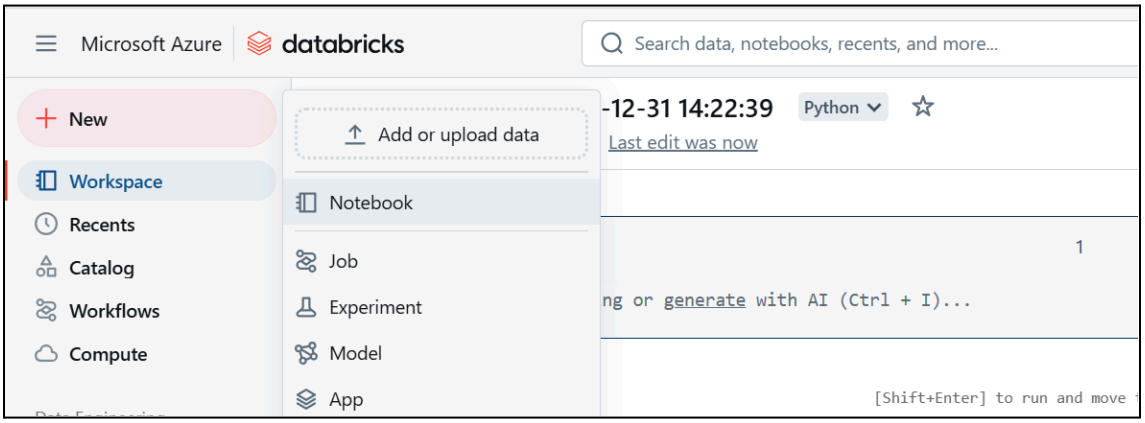


g) In the Databricks interface, the left panel includes the following sections:

- **Workspace**
- **Compute**
- **Job Runs**
- **Catalog**

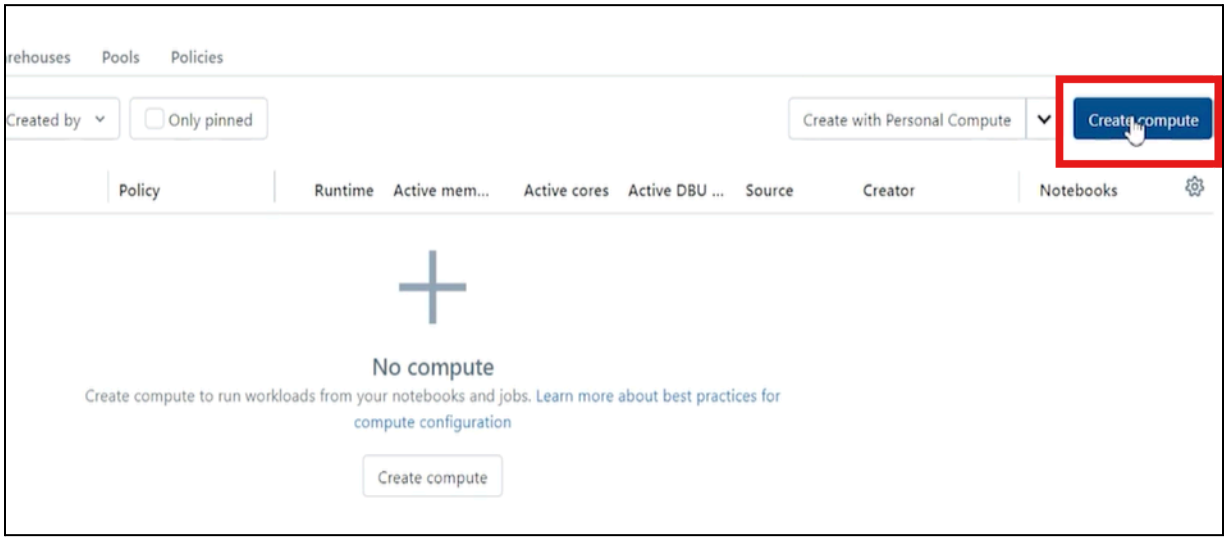


h) Firstly, a cluster will be created inside Databricks Workspace. In the left side panel, click on **Compute.**

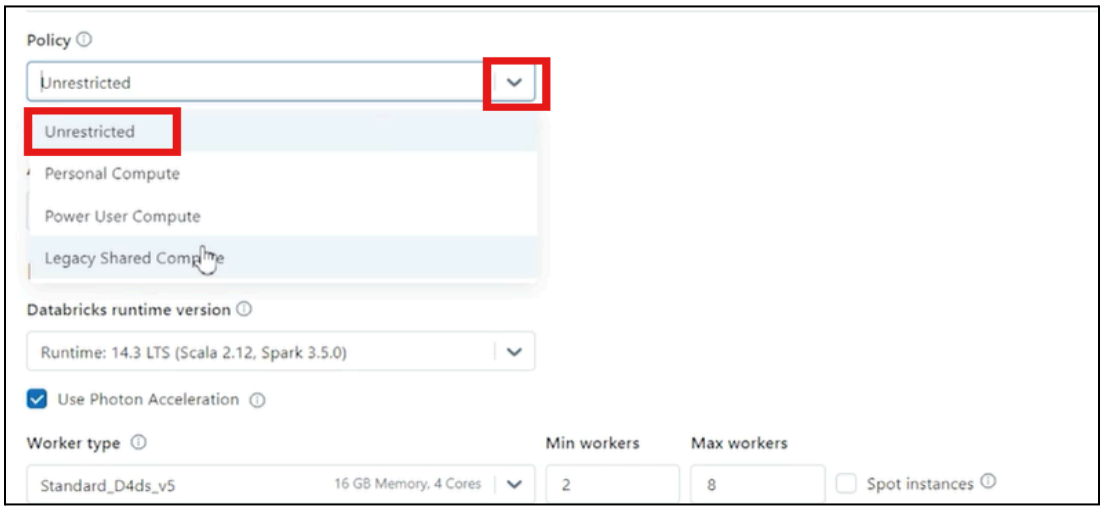


i) Click on **Create Compute.**

A cluster is necessary for Databricks to run code and process data during the running of jobs, as it provides the computational resources.

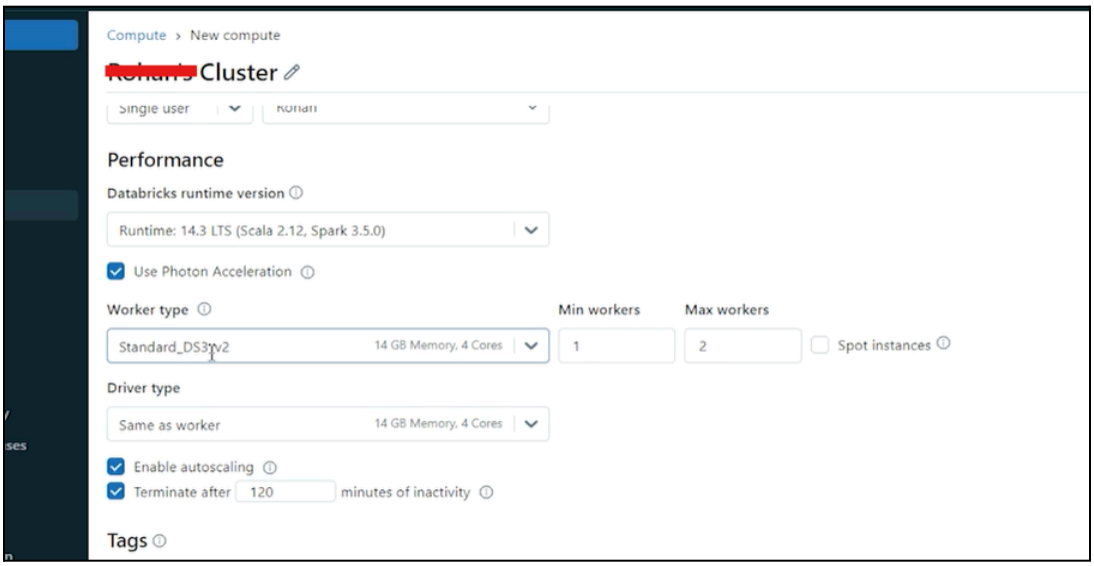


j) In the policy, from the dropdown - select **“unrestricted” or** go with **“Personal Compute”** if you face any error.

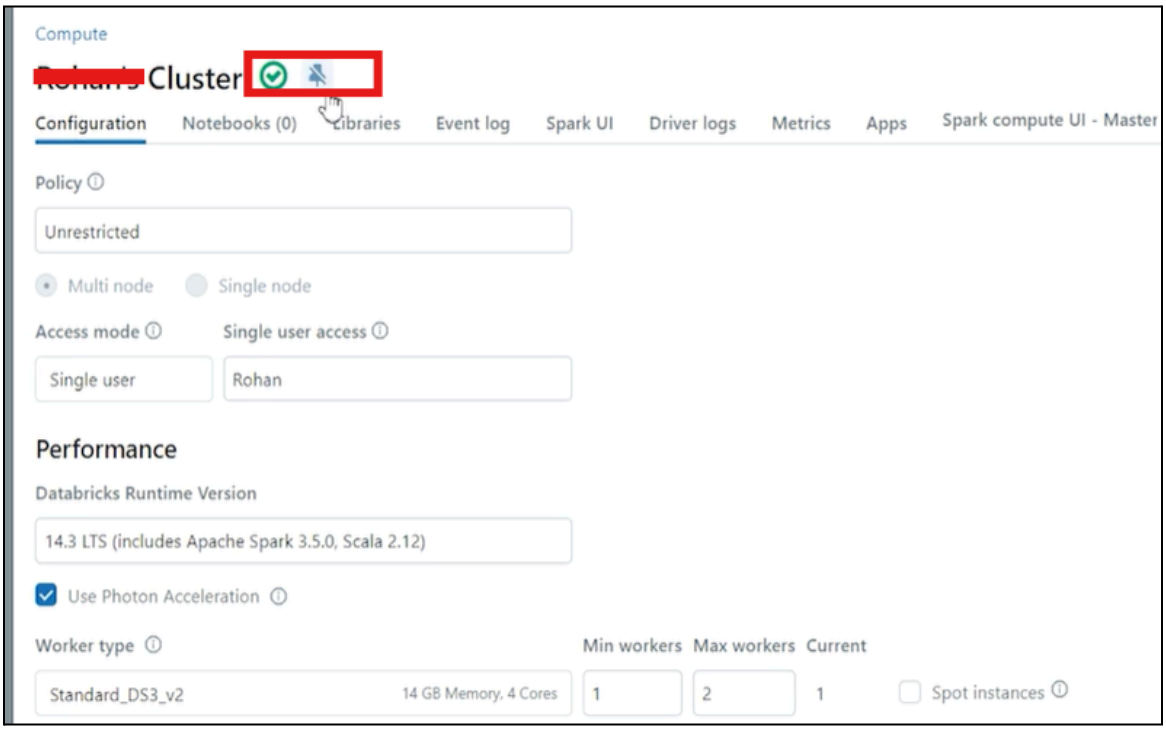


k) Keep all other configurations as shown in the screenshot; this will **reduce costs and save compute resources.**

Click on **“Create Compute”** to create the cluster.



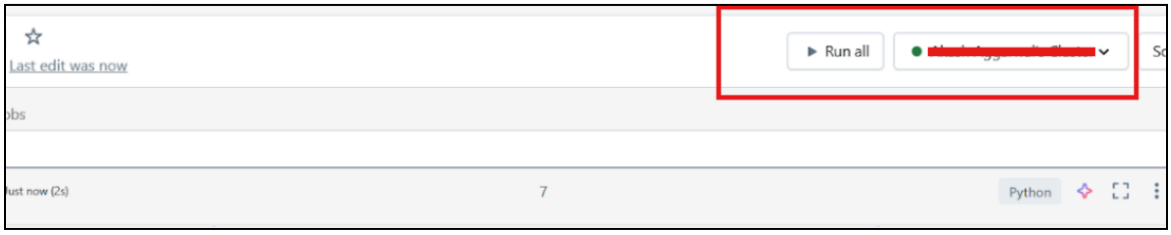
The cluster may take some time to create. Once it is ready, the **right signal** will appear next to the cluster name.



Uploading Notebook:

Once everything is configured, such as the Databricks workspace setup and the cluster provisioning, the next step is to upload the **project notebook to the Databricks workspace**. This notebook contains the PySpark code necessary for processing and analyzing the Yelp dataset.

- Download the code folder for this project.
- Unzip the code folder ZIP file on your local system.
- Click on **Workspace** in the Databricks UI, then click on **Notebook**.
- Click on the **Upload** button to upload the **notebook file(yelp-dataset-analysis)** from your local system into the Databricks workspace.
- Open the uploaded notebook and ensure that the cluster you created is attached to it, as the cluster serves as the compute resource responsible for running the Spark job within the Databricks environment.



Execution (Spark Job):

Now, the code notebook needs to be executed cell by cell within the Databricks environment using the Spark job. Each code cell is designed with detailed explanations, so you can easily follow along and understand the logic behind each step.

Simply run the cells sequentially from start to finish to perform the complete data processing and analysis on the Yelp dataset.

Note: *Since mounting has been deprecated in the Databricks environment, it is no longer recommended by Databricks. Therefore, instead of mounting the containers created inside the Azure Blob Storage account and setting up a mount point, this project will directly access the storage containers and their subfolders to retrieve data and perform replication.*

In the code folder, we have specified direct access to the cloud object storage by specifying details such as the storage account name and access keys. This is the simplest way to authenticate and connect to ADLS Gen2 within the Databricks workspace.

However, if you prefer not to use credentials directly in ***Azure Databricks****, you can create* ***Secrets and Scopes*** *inside* ***Azure Databricks****. This method allows you to securely store credentials within the scopes and access them inside the Databricks environment without exposing sensitive information.*

For simplicity, in this project, we have opted to specify the credentials directly in the Databricks notebooks. While this is the easiest approach, you can also choose to use Secrets for better security.

Code Explanation:

1. Importing Required PySpark Modules for Data Transformation

```
# Import packages

import pyspark.sql.functions as f
from pyspark.sql.window import Window
```


```
# Import packages
import pyspark.sql.functions as f
from pyspark.sql.window import Window
```


Explanation:

- **import pyspark.sql.functions as f:** This line imports all commonly used PySpark SQL functions and aliases them as f for easier access. These include functions like col(), when(), count(), avg(), lit(), and many more that are essential for DataFrame transformations, aggregations, filtering, and calculations.

- **from pyspark.sql.window import Window: T**his line imports the Window class, which allows operations like ranking, running totals, or row-wise comparisons to be performed across partitions of data (similar to SQL window functions). This is useful for analytical operations like identifying top-N records or computing cumulative statistics within groups.

This code snippet sets up the required PySpark modules needed for performing data transformations and advanced analytical queries.

The **functions module** provides tools for column-wise operations, and the Window module enables complex partition-based analytics, both of which are foundational for processing structured datasets in Spark.

2. Setting Up Authentication for Azure Data Lake Storage in Spark

```
spark.conf.set("fs.azure.account.key.sgyelp.dfs.core.windows.net",
"<access-key>")
```


```
spark.conf.set("fs.azure.account.key.sgyelp.dfs.core.windows.net
", "<access-key>")
```

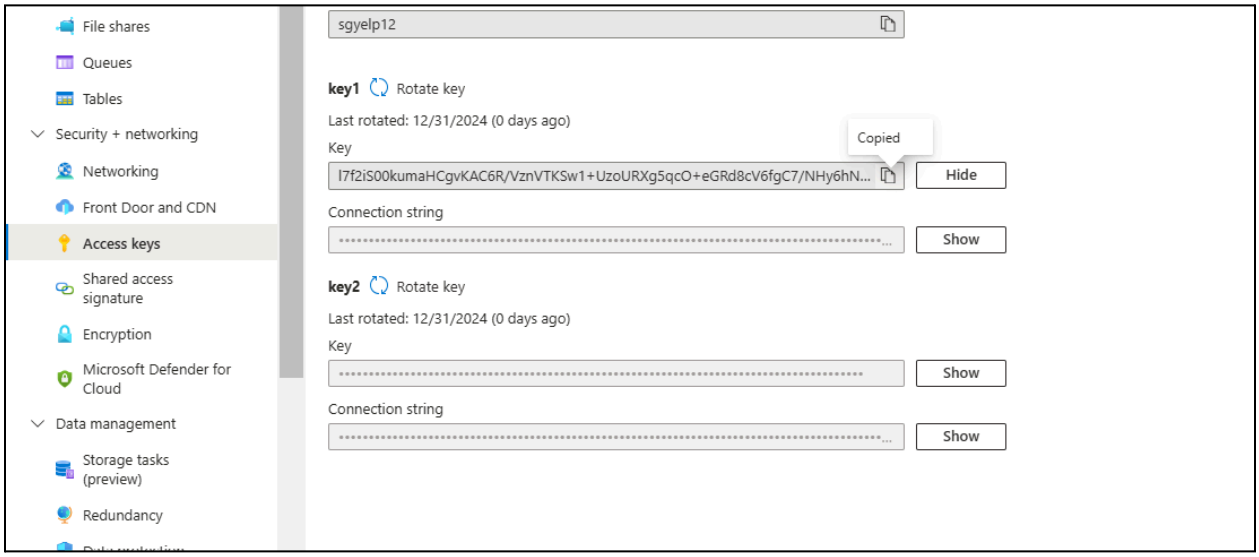

Remember to replace **<access-key>** with your actual key before running the code.

Explanation:

This line configures Spark to authenticate with Azure Data Lake Storage (ADLS) Gen2 using the storage account access key. The key allows the Databricks Spark environment to securely read from and write to the specified ADLS account (sgyelp.dfs.core.windows.net). This is essential when you're connecting Databricks to external storage resources where the raw and processed data is kept.

Make sure to replace **"<access-key>"** with the actual access key for your Azure storage account. Without this key, Spark will not be able to access the data stored in your ADLS containers.

- Go to the Azure Portal
- Navigate to your Storage Account (e.g., sgyelp)
- Click on Access keys under **Security + Networking**



- Choose one of the two available keys **(key1 or key2)**
- Click the Copy icon to copy the key to your clipboard
- Replace the placeholder **<access-key>** in the code with your copied key

This code snippet authenticates your Databricks Spark session with Azure Data Lake Storage Gen2 by setting the storage account key in the Spark configuration. This enables Spark to access data stored in your ADLS containers, such as reading **Yelp JSON files** or writing transformed data.

3. Listing Files in the Source Container in Azure Data Lake Storage (ADLS)

```
dbutils.fs.ls("abfss://<your-container-name>@<your-storage-account-name>.dfs.core.windows.net/")
```

dbutils.fs.ls("abfss://<your-container-name>@<your-storage-account-name>.dfs.core.windows.net/")

To make this work for your setup, you need to **replace placeholders**:

- Replace sourcecontainer with the **actual name of your destination container** (the one used as the source in Databricks, where ADF copied the files).
- Replace sgyelp with your **actual ADLS Gen2 storage account name**.

This line lists all files and directories located in the specified container of your **Azure Data Lake Storage Gen2 (ADLS)** using the ABFSS (Azure Blob File System Secure) protocol. It uses dbutils.fs.ls() to query the contents of the **sourcecontainer** within the storage account named sgyelp.

This code snippet helps you confirm that the **Yelp JSON files** were successfully copied to your ADLS Gen2 container by listing its contents. Just ensure to substitute the container and storage account placeholders with your actual names before running.

Note: abfss Protocol

*The **abfss protocol** stands for Azure Blob File System Secure, and it is a secure extension of the **abfss protocol** used to access Azure Data Lake Storage Gen2 (ADLS Gen2).*

Why use abfss://?

- It ensures **encrypted communication** between your Spark environment (like Databricks) and the Azure storage.
- It is the **recommended protocol** for production workloads due to security best practices.

4. Conversion of JSON to Parquet Format

As discussed earlier, **Parquet is an optimized columnar storage file format** that is highly efficient for both storage and processing, especially in big data environments like Apache Spark. In this step, we read the Yelp JSON files into individual Spark DataFrames, then convert and write each one into Parquet format inside a dedicated directory **within the Azure Data Lake Storage Gen2 (ADLS Gen2) container.**

This transformation allows faster querying, reduced storage size, and better performance in downstream processing.

```
df_business =
spark.read.json("abfss://<your-container-name>@<your-storage-accoun
t-name>.dfs.core.windows.net/yelp_academic_dataset_business.json")

df_business.write.mode('overwrite').parquet("abfss://<your-container-n
ame>@<your-storage-account-name>.dfs.core.windows.net/json_to_par
quet/business.parquet")



df_checkin =
spark.read.json("abfss://<your-container-name>@<your-storage-accoun
t-name>.dfs.core.windows.net/yelp_academic_dataset_checkin.json")

df_checkin.write.mode('overwrite').parquet("abfss://<your-container-na
me>@<your-storage-account-name>.dfs.core.windows.net/json_to_par
quet/checkin.parquet")



df_review =
spark.read.json("abfss://<your-container-name>@<your-storage-accoun
t-name>.dfs.core.windows.net/yelp_academic_dataset_review.json")

df_review.write.mode('overwrite').parquet("abfss://<your-container-nam
e>@<your-storage-account-name>.dfs.core.windows.net/json_to_parqu
```

```
df_business =
spark.read.json("abfss://<your-container-name>@<your-storage-acc
ount-name>.dfs.core.windows.net/yelp_academic_dataset_business.j
son")
df_business.write.mode('overwrite').parquet("abfss://<your-conta
iner-name>@<your-storage-account-name>.dfs.core.windows.net/json
_to_parquet/business.parquet")
```

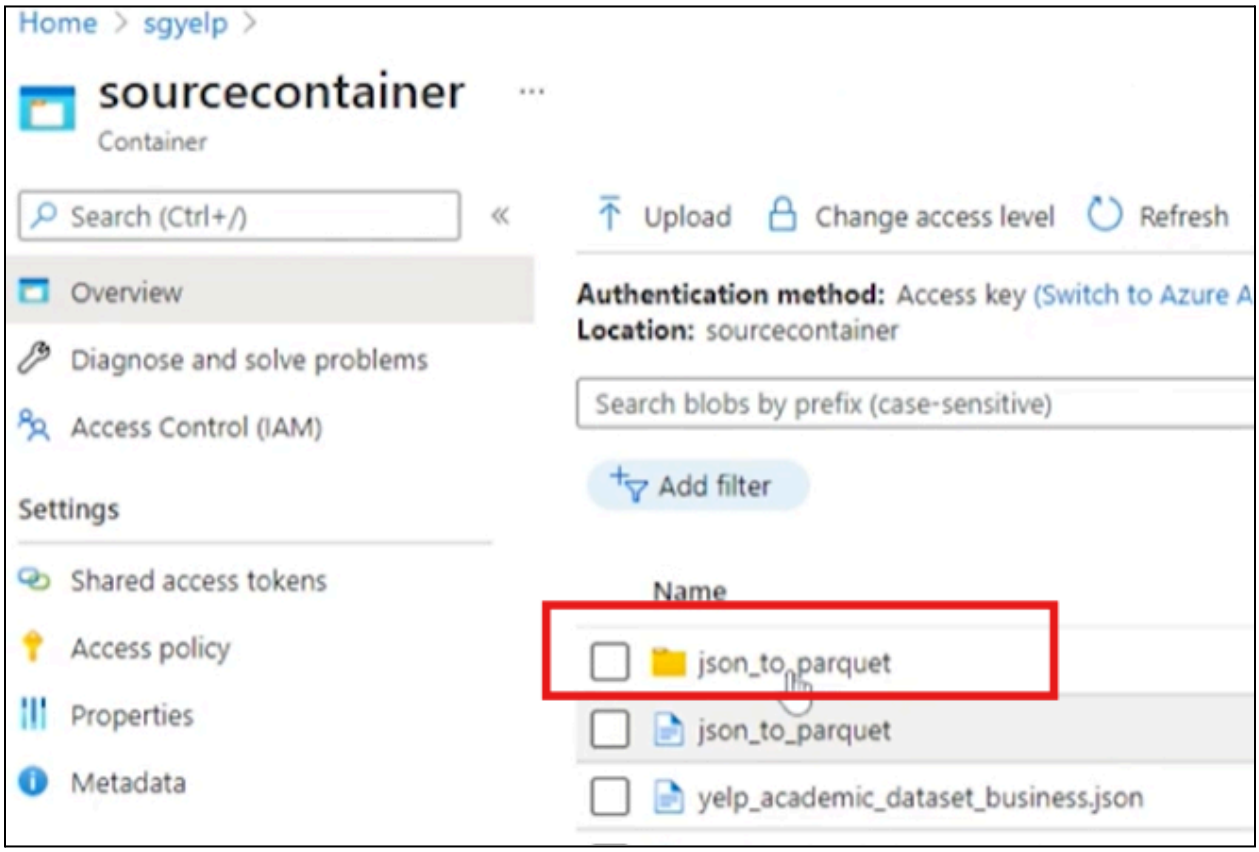
```
df_checkin =
spark.read.json("abfss://<your-container-name>@<your-storage-acc
ount-name>.dfs.core.windows.net/yelp_academic_dataset_checkin.js
on")
df_checkin.write.mode('overwrite').parquet("abfss://<your-contai
ner-name>@<your-storage-account-name>.dfs.core.windows.net/json_
to_parquet/checkin.parquet")

df_review =
spark.read.json("abfss://<your-container-name>@<your-storage-acc
ount-name>.dfs.core.windows.net/yelp_academic_dataset_review.jso
n")
df_review.write.mode('overwrite').parquet("abfss://<your-contain
er-name>@<your-storage-account-name>.dfs.core.windows.net/json_t
o_parquet/review.parquet")

df_tip =
spark.read.json("abfss://<your-container-name>@<your-storage-acc
ount-name>.dfs.core.windows.net/yelp_academic_dataset_tip.json")
df_tip.write.mode('overwrite').parquet("abfss://<your-container-
name>@<your-storage-account-name>.dfs.core.windows.net/json_to_p
arquet/tip.parquet")

df_user =
spark.read.json("abfss://<your-container-name>@<your-storage-acc
ount-name>.dfs.core.windows.net/yelp_academic_dataset_user.json"
)
df_user.write.mode('overwrite').parquet("abfss://<your-container
-name>@<your-storage-account-name>.dfs.core.windows.net/json_to_
parquet/user.parquet")
```

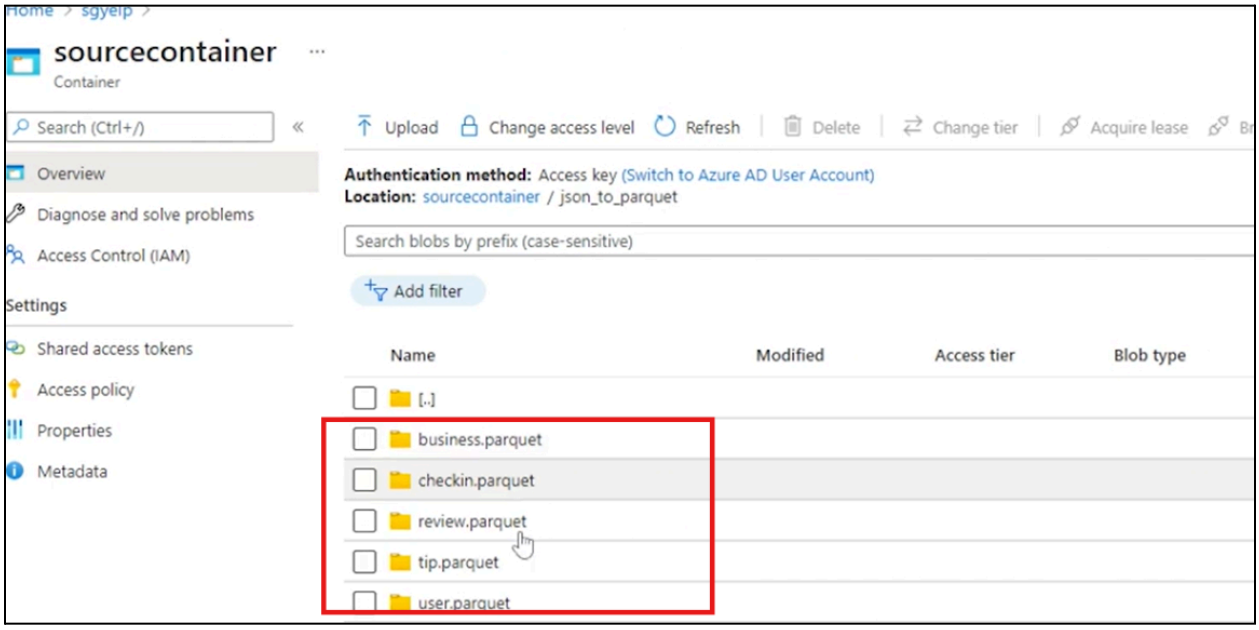
Make sure to replace the **container name and the Azure Data Lake Storage account name** in the code with your actual container name and storage account name.

This code reads all five Yelp JSON datasets — **business, checkin, review, tip, and user** — using **Spark's .read.json()** method. It then writes each DataFrame to the ADLS container using the **.write.mode('overwrite').parquet()** method, storing them in the json_to_parquet directory for organized access.

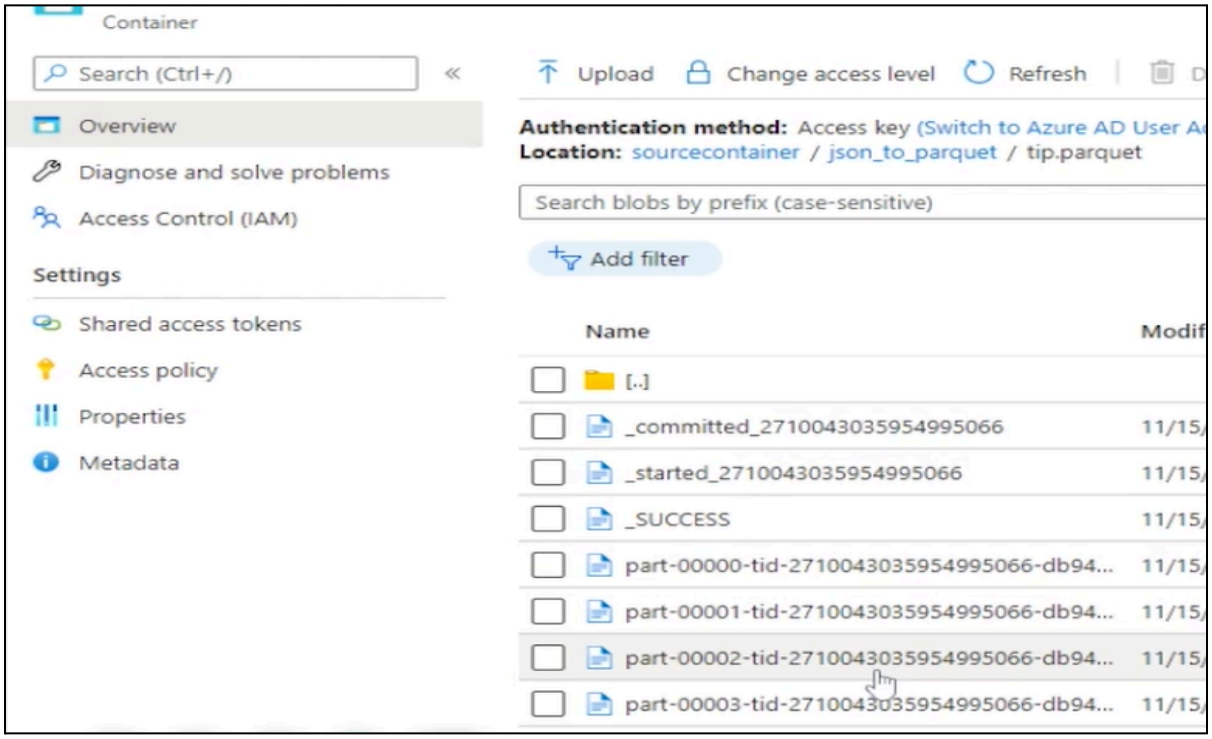
This code snippet converts raw Yelp JSON files to the Parquet format and saves them efficiently in the Azure Data Lake Storage Gen2 environment, enabling faster and scalable data processing for analytical tasks.

- Go to the **Azure Data Lake Storage container (sourcecontainer).**
- You will see the Parquet files of different Yelp datasets, as the code that was run converted the JSON data to the Parquet format.





When you open any folders, you will see the parquet files like that:



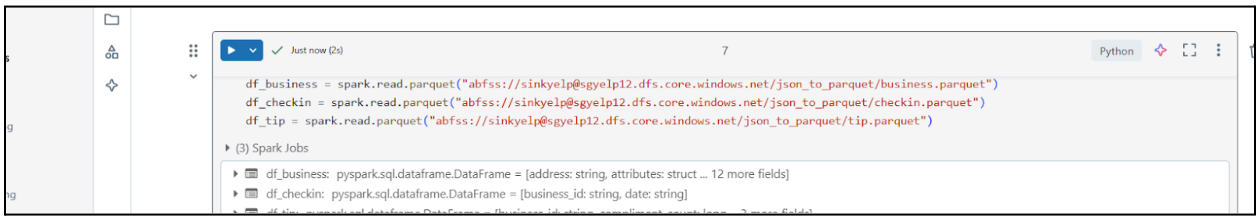
5. Reading Parquet Files into DataFrames

Now that all the Yelp datasets have been successfully converted and stored in **Parquet format** inside the Azure Data Lake Storage Gen2 container, the next step is to **read these Parquet files back into Spark DataFrames**. This enables us to perform further analysis in an optimized and scalable manner, leveraging Spark's distributed processing power.

Parquet files are columnar, which means reading only specific columns is faster and more efficient than working with **row-based formats like JSON.** This step ensures that any downstream analytics, filtering, and transformations will be faster and more cost-effective.

```
df_business =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/json_to_parquet/business.parquet")

df_checkin =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/json_to_parquet/checkin.parquet")

df_review =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/json_to_parquet/review.parquet")

df_tip =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/json_to_parquet/tip.parquet")

df_user =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/json_to_parquet/user.parquet")
```

```
df_business =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.wind
ows.net/json_to_parquet/business.parquet")
df_checkin =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.wind
ows.net/json_to_parquet/checkin.parquet")
df_review =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.wind
ows.net/json_to_parquet/review.parquet")
df_tip =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.wind
ows.net/json_to_parquet/tip.parquet")
df_user =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.wind
ows.net/json_to_parquet/user.parquet")
```



Each line in the code above reads a single Parquet file from the json_to_parquet/ directory in **the Azure Data Lake Storage Container(sourcecontainer)** and loads it into a corresponding DataFrame. These DataFrames now contain structured, optimized data, ready for querying, transformation, and analysis using PySpark.

This code snippet efficiently loads the previously stored Parquet files into Spark DataFrames. These DataFrames are essential for enabling fast, distributed analysis of the Yelp dataset in Databricks.

6. Counting Rows in Each DataFrame

```
print("df_tip: " + str(df_tip.count()))

print("df_checkin: " + str(df_checkin.count()))

print("df_business: " + str(df_business.count()))

print("df_review: " + str(df_review.count()))

print("df_user: " + str(df_user.count()))
```

This code block is used to **verify the number of records** in each DataFrame that was read from the Parquet files. Counting rows is a useful way to ensure that the data has been successfully loaded and to understand the volume of data for each dataset.

Explanation:

- .count() is an **action** in PySpark that triggers the computation and returns the total number of rows in the DataFrame.
- str() converts the numeric result of .count() into a string so it can be concatenated and printed in a readable format.
- Each line prints the name of the dataset and the corresponding record count.

This code snippet checks each dataset's total number of rows after loading them from Parquet files. It's a quick validation step to ensure the data is properly loaded. It gives an idea of the dataset sizes for tip, check-in, business, review, and user information in the Yelp dataset.

7. Displaying the Tip DataFrame

This code cell is intended to **visually inspect the contents of the `df_tip` DataFrame** in the Databricks notebook environment.


```
display(df_tip)
```

Explanation:

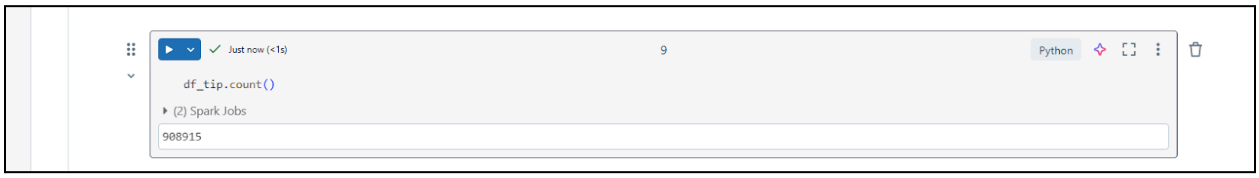
- **display()** is a **Databricks-specific function** used to render DataFrames in a **tabular, interactive format** within notebooks.
- It provides a user-friendly interface with features like column sorting, filtering, and pagination, making it easier to **explore the data** without writing additional code.

This code snippet displays the content of the df_tip DataFrame in a structured, visual format within the Databricks notebook UI. It helps perform a quick data inspection to understand the structure and values of the Yelp tip dataset before applying any transformations or analysis.

8. Counting Records in the Tip DataFrame

This code cell is used to **count the total number of rows** (records) present in the **df_tip DataFrame.**


```
df_tip.count()
```



Explanation:

- The **.count()** function is a **Spark DataFrame action** that **triggers execution** and returns the total number of rows in the DataFrame.
- Since Spark uses lazy evaluation, this operation will **scan the entire DataFrame**, making it a good checkpoint to understand the data size before processing.
- df_tip refers to the DataFrame that was read from the Yelp tip dataset stored in Parquet format on Azure Data Lake Storage.

This code snippet returns the total number of tip records available in the `df_tip` DataFrame, helping you understand the dataset's size before proceeding with further transformations or analysis.

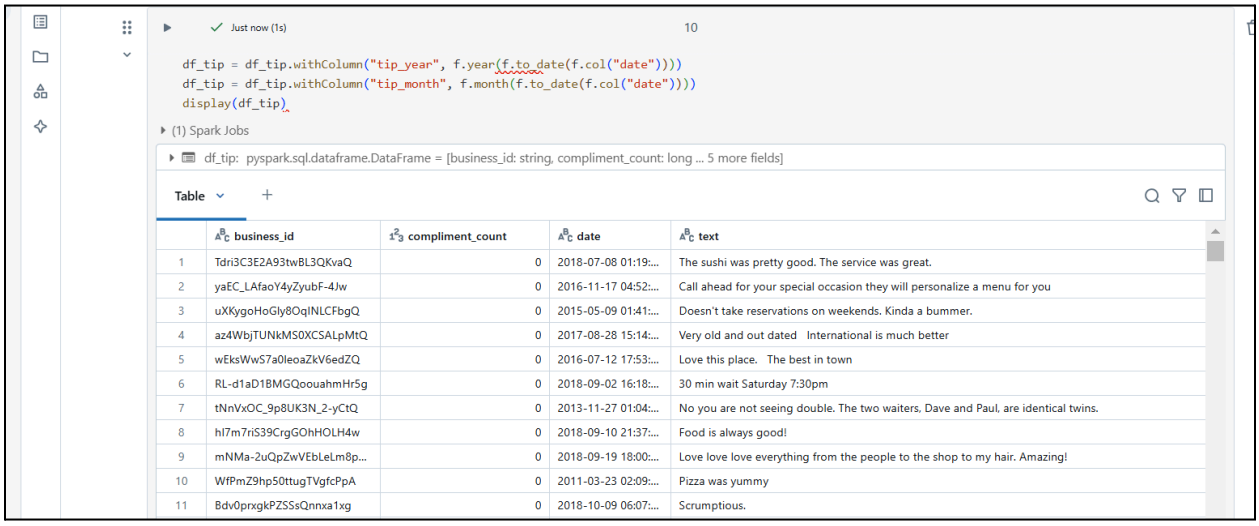
9. Extracting Year and Month from Tip Date

This code cell is designed to **add two new columns** to the `df_tip` DataFrame: one for the **year** and another for the **month** extracted from the `date` column in the dataset.

```
df_tip = df_tip.withColumn("tip_year",f.year(f.to_date(f.col("date"))))
df_tip = df_tip.withColumn("tip_month", f.month(f.to_date(f.col("date"))))
display(df_tip)
```

Code:

```
df_tip = df_tip.withColumn("tip_year",
f.year(f.to_date(f.col("date"))))
df_tip = df_tip.withColumn("tip_month",
f.month(f.to_date(f.col("date"))))
display(df_tip)
```



Just now (1s) 10

```
df_tip = df_tip.withColumn("tip_year", f.year(f.to_date(f.col("date"))))
df_tip = df_tip.withColumn("tip_month", f.month(f.to_date(f.col("date"))))
display(df_tip)
```

(1) Spark Jobs

df_tip: pyspark.sql.dataframe.DataFrame = [business_id: string, compliment_count: long ... 5 more fields]

Table

	business_id	compliment_count	date	text
1	Tdri3C3E2A93twBL3QKvaQ	0	2018-07-08 01:19:...	The sushi was pretty good. The service was great.
2	yaEC_LAfaoY4yZyubF-4Jw	0	2016-11-17 04:52:...	Call ahead for your special occasion they will personalize a menu for you
3	uXKygoHoGly8OqINLCFbgQ	0	2015-05-09 01:41:...	Doesn't take reservations on weekends. Kinda a bummer.
4	az4WbjTUNkMS0XCSALpMtQ	0	2017-08-28 15:14:...	Very old and out dated International is much better
5	wEksWwS7a0leoaZkV6edZQ	0	2016-07-12 17:53:...	Love this place. The best in town
6	RL-d1aD1BMGQoouahmHr5g	0	2018-09-02 16:18:...	30 min wait Saturday 7:30pm
7	tNnVxOC_9p8UK3N_2-yCtQ	0	2013-11-27 01:04:...	No you are not seeing double. The two waiters, Dave and Paul, are identical twins.
8	hI7m7riS39CrgGOhHOLH4w	0	2018-09-10 21:37:...	Food is always good!
9	mNMa-2uQpZwVEbLeLm8p...	0	2018-09-19 18:00:...	Love love love everything from the people to the shop to my hair. Amazing!
10	WfPmZ9hp50ttugTVgfcPpA	0	2011-03-23 02:09:...	Pizza was yummy
11	Bdv0prxgkPZSsQnnxa1xg	0	2018-10-09 06:07:...	Scrumptious.

Explanation:

- f.col("date"): Refers to the original date column in the df_tip DataFrame.
- f.to_date(...): Converts the date column (likely a string) into a proper DateType format so it can be used for date/time operations.
- f.year(...): Extracts the year from the converted date and creates a new column named tip_year.
- f.month(...): Extracts the month and stores it in a new column named tip_month.

- withColumn(...): Used to add or replace columns in the DataFrame.
- display(df_tip): Displays the updated DataFrame to visually confirm that the new columns were added successfully.

This code snippet transforms the tip dataset by adding two additional columns—tip_year and tip_month—based on the existing date column. This enhancement allows for **temporal analysis** of tips, such as trends over time or seasonality in tip frequency.

10. Partitioning and Optimizing Data with Parquet in Spark


```
df_tip.write.mode("overwrite") \
    .partitionBy("tip_year", "tip_month") \

.parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/tip_partitioned_by_year_and_month/")
```


```
df_tip.write.mode("overwrite") \
    .partitionBy("tip_year", "tip_month") \

.parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/tip_partitioned_by_year_and_month/")
```

Explanation:

- df_tip.write: Initiates the DataFrame write operation.
- .mode("overwrite"): Ensures that if the output path already exists, it will be replaced with the new data.
- .partitionBy("tip_year", "tip_month"): Physically splits and stores the data in subdirectories based on tip_year and tip_month columns. This greatly improves query performance during read operations, especially when filtering by these fields.
- .parquet(...): Specifies the output format as Parquet, a columnar and compressed file format that is optimal for big data processing.
- "abfss://...": This is the Azure Data Lake Storage (ADLS) Gen2 path using the ABFSS protocol.

Replace:

  - sourcecontainer with your container name
  - sgyelp with your Azure Data Lake Storage account name

This code partitions the df_tip DataFrame by tip_year and tip_month and writes it in Parquet format to Azure Data Lake Storage. Partitioning organizes the dataset into year- and month-based folders, improving query speed and efficiency during data retrieval and further Spark transformations.

ProjectPro

11. Partition Control Using Coalesce and Repartition in Spark

```
print("current number of partitions: " + str(df_user.rdd.getNumPartitions()))

df_reduce_part = df_user.coalesce(10)
print("reduced number of partitions after coalesce: " +
str(df_reduce_part.rdd.getNumPartitions()))

df_increased_df = df_user.repartition(30)
print("increased number of partitions after coalesce: " +
str(df_increased_df.rdd.getNumPartitions()))
```


```
print("current number of partitions: " +
str(df_user.rdd.getNumPartitions()))

df_reduce_part = df_user.coalesce(10)
print("reduced number of partitions after coalesce: " +
str(df_reduce_part.rdd.getNumPartitions()))

df_increased_df = df_user.repartition(30)
print("increased number of partitions after coalesce: " +
str(df_increased_df.rdd.getNumPartitions()))
```

Explanation:

- df_user.rdd.getNumPartitions(): Returns the number of partitions currently present in the df_user DataFrame's underlying RDD (Resilient Distributed Dataset).

- df_user.coalesce(10): Reduces the number of partitions to 10 using coalesce, which merges adjacent partitions without full data shuffle — it's efficient and used to reduce partitions.

- df_user.repartition(30): Increases the number of partitions to 30 using repartition, which performs a full shuffle of the data. This is useful when you want to increase parallelism in processing.

This code helps analyze and control partitioning in the Spark environment for the df_user DataFrame. It first checks the existing number of partitions, then demonstrates how to reduce partitions using coalesce and increase them using repartition. Controlling partitions is key for performance tuning in distributed data processing with Spark.

12. Coalescing and Writing Data to Parquet Format

```
df_user.coalesce(10).write.mode('overwrite').parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/coalesce/user.parquet")'
```

df_user.coalesce(10).write.mode('overwrite').parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/coalesce/user.parquet")'

Explanation:

- df_user.coalesce(10): This reduces the number of partitions in the df_user DataFrame to 10 before writing. Coalescing minimizes the number of output files by combining partitions, which is particularly useful when writing to storage systems like Azure Data Lake Storage Gen2.

- .write.mode('overwrite'): Ensures that if a file or folder already exists at the target path, it will be overwritten.

- .parquet(...): Specifies the output format for the data as Parquet, a columnar storage format optimized for performance and efficient querying.

- "abfss://sourcecontainer@sgyelp.dfs.core.windows.net/coalesce/user.parquet":

    - abfss is the secure protocol for accessing Azure Data Lake Storage Gen2.

    - Replace "sourcecontainer" with your actual container name.

    - Replace "sgyelp" with your actual Data Lake Storage account name.

This code snippet optimizes the write operation by reducing the number of output partitions (and files) using coalesce before saving the df_user DataFrame as a Parquet file in Azure Data Lake Storage. It improves manageability and potentially enhances performance when reading the data later for further processing or analysis.

13. Writing Parquet File Using Repartition in Spark

```
df_user.repartition(10).write.mode('overwrite').parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/repartition/user.parquet")
```

df_user.repartition(10).write.mode('overwrite').parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/repartition/user.parquet")

Explanation:

- df_user.repartition(10): This increases or redistributes the data into 10 partitions. Unlike coalesce(), which only reduces partitions, repartition() can both increase and redistribute data across partitions, ensuring a more uniform data distribution.
- .write.mode('overwrite'): Specifies that if there is any existing data at the target location, it will be overwritten.
- .parquet(...): Writes the repartitioned DataFrame in Parquet format, which supports efficient columnar storage and query performance.
- "abfss://sourcecontainer@sgyelp.dfs.core.windows.net/repartition/user.parquet":

- abfss: Azure Blob File System Secure protocol.
- Replace "sourcecontainer" with your container name.
- Replace "sgyelp" with your actual Azure Data Lake Storage account name.

This code snippet demonstrates how to use repartition() to evenly redistribute the df_user DataFrame into 10 partitions before writing it as a Parquet file in Azure Data Lake Storage. Repartitioning can help optimize performance during large-scale reads and distributed processing, especially when data skew is a concern.

14. Reading Repartitioned Parquet File into DataFrame

```
df_user =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.windows.net/repartitio
n/user.parquet")
display(df_user)
```

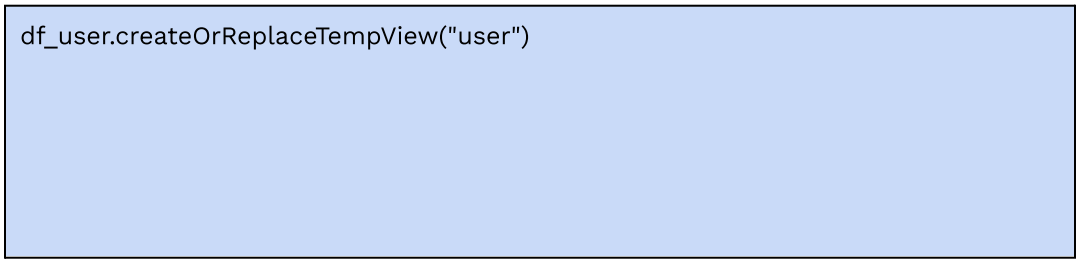

```
df_user =
spark.read.parquet("abfss://sourcecontainer@sgyelp.dfs.core.wind
ows.net/repartition/user.parquet")
display(df_user)
```

Explanation:

- spark.read.parquet(...): This reads a Parquet file from Azure Data Lake Storage Gen2 into a Spark DataFrame. It is efficient due to Parquet's columnar format.
- "abfss://sourcecontainer@sgyelp.dfs.core.windows.net/repartition/user.parquet":
    - abfss: Secure protocol for accessing Azure Data Lake Storage.
    - Replace "sourcecontainer" with your actual container name.
    - Replace "sgyelp" with your storage account name if different.
- display(df_user): A Databricks-specific function that visually displays the DataFrame content in a tabular form, which is helpful for quickly validating and inspecting data.

This code snippet loads the repartitioned user.parquet file from ADLS into a Spark DataFrame and displays it. It's a verification step to ensure the repartitioned file was successfully written and is readable, allowing you to continue analysis or further transformations.

15. Creating a Temporary SQL View for Spark SQL Queries




```
df_user.createOrReplaceTempView("user")
```

Explanation:

- df_user: This is the Spark DataFrame containing user-related data, which has been previously loaded from a Parquet file.
- .createOrReplaceTempView("user"):
    - This registers the df_user DataFrame as a temporary SQL view named "user".
    - This view can now be queried using Spark SQL just like a SQL table.
    - createOrReplaceTempView ensures that if a view named "user" already exists, it will be replaced with this one.

This line of code enables you to run SQL queries on the user DataFrame by creating a temporary view named "user" within the Spark session. This makes it easier to interact with the data using familiar SQL syntax without needing to refer to the DataFrame directly in every operation.

16. Top 3 Users by Review Count

```
SELECT
  user_id,
  name,
  review_count
FROM
  user
ORDER BY
  review_count DESC -- highest review_count users will be listed first as it is a
descending order
LIMIT
  3;
```

```
SELECT
  user_id,
  name,
  review_count
FROM
  user
ORDER BY
  review_count DESC -- highest review_count users will be listed
first as it is a descending order
LIMIT
  3;
```

Explanation:

- This is a Spark SQL query executed against the temporary view named user (which we created earlier from the df_user DataFrame).
- SELECT user_id, name, review_count: Selects the columns of interest — the user's unique ID, their name, and the number of reviews they've posted.
- ORDER BY review_count DESC: Sorts the records in descending order based on the review_count column, meaning the users who have written the most reviews will appear at the top.
- LIMIT 3: Limits the output to just the top 3 users with the highest number of reviews.

This query retrieves the top 3 most active users based on the number of reviews they've written. By sorting in descending order and limiting to three rows, it quickly surfaces the most prolific contributors in the dataset.

17. Top 10 Users by Number of Fans

```
SELECT
  user_id,
  name,
  fans
FROM
  user
ORDER BY
  fans DESC -- arrange user_id in descending order of number of fans
LIMIT
  10 -- limit to first 10 rows
```


```
SELECT
  user_id,
  name,
  fans
FROM
  user
ORDER BY
  fans DESC -- arrange user_id in descending order of number of
fans
LIMIT
  10 -- limit to first 10 rows
```

Explanation:

- This SQL query fetches data from the user's temporary view in Databricks.
- SELECT user_id, name, fans: Retrieves each user's ID, name, and the number of fans they have.
- ORDER BY fans DESC: Sorts the records in descending order based on the number of fans. The users with the most fans appear first.
- LIMIT 10: Limits the output to only the top 10 users based on fan count.

This code snippet extracts the 10 most popular users in the Yelp dataset, determined by how many fans they have. It helps highlight the most influential users within the community, potentially useful for identifying key reviewers or brand ambassadors.

18. Display Business DataFrame

```
SELECT
  user_id,
  name,
  fans
FROM
  user
ORDER BY
  fans DESC -- arrange user_id in descending order of number of fans
LIMIT
  10 -- limit to first 10 rows
```

```
display(df_business)
```

Explanation:

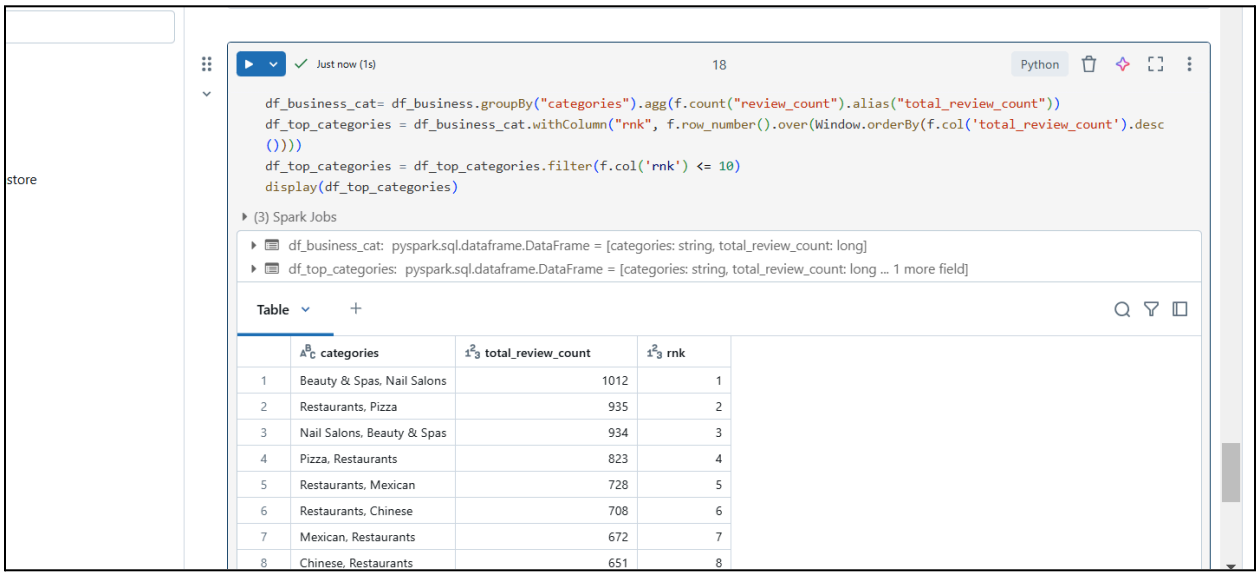
- This command is used in Databricks notebooks to visually display the contents of the `df_business` DataFrame in tabular format.
- The `display()` function is a Databricks-specific feature that allows rich visualization directly in the notebook, such as sorting, filtering, and even plotting graphs from the DataFrame.
- Here, `df_business` contains data loaded from the Yelp Business dataset, which includes details like business ID, name, address, location, categories, ratings, etc.

This code snippet visually presents the Yelp Business dataset in an interactive table within the Databricks notebook. It allows you to manually explore the records and understand the structure and content of the business data.

19. Top 10 Business Categories by Review Count

```
df_business_cat =
df_business.groupBy("categories").agg(f.count("review_count").alias("total_review_co
unt"))
df_top_categories = df_business_cat.withColumn("rnk",
f.row_number().over(Window.orderBy(f.col('total_review_count').desc())))
df_top_categories = df_top_categories.filter(f.col('rnk') <= 10)
display(df_top_categories)
```

```
df_business_cat =
df_business.groupBy("categories").agg(f.count("review_count").al
ias("total_review_count"))
df_top_categories = df_business_cat.withColumn("rnk",
f.row_number().over(Window.orderBy(f.col('total_review_count').d
esc())))
df_top_categories = df_top_categories.filter(f.col('rnk') <= 10)
display(df_top_categories)
```



Explanation:

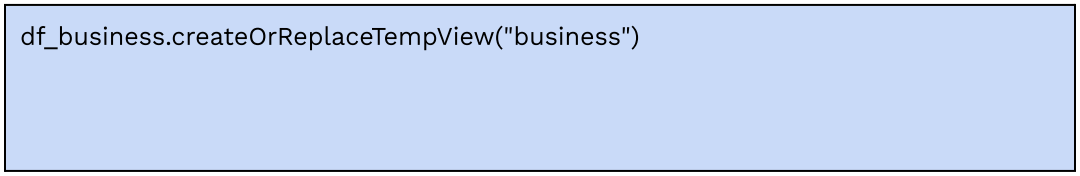
- df_business.groupBy("categories"): Groups the business dataset by the categories column.

- .agg(f.count("review_count").alias("total_review_count")): For each category group, it counts the number of review entries and renames it to total_review_count.

- withColumn("rnk", ...): Adds a new column rnk using row_number() window function to assign a rank based on descending order of total review counts.
- Window.orderBy(f.col('total_review_count').desc()): The window spec ensures the highest reviewed categories come first.
- .filter(f.col('rnk') <= 10): Filters to retain only the top 10 ranked categories.
- display(df_top_categories): Displays the result in a rich interactive table format.

This code snippet identifies and ranks the top 10 business categories based on the total number of reviews in the Yelp dataset. It helps highlight which business types are most engaged with by users, offering a strong signal of popularity or customer interest.

20. Registering Business DataFrame as a Temporary View




```
df_business.createOrReplaceTempView("business")
```

Explanation:

- This line registers the df_business DataFrame as a temporary view named "business" in the Spark SQL context.
- Once registered, you can query the data using SQL syntax directly within your Databricks notebook or script.

- createOrReplaceTempView() ensures that if a view with the same name already exists, it will be replaced with the new one.

- This temporary view exists only for the duration of the Spark session and is not persisted permanently.

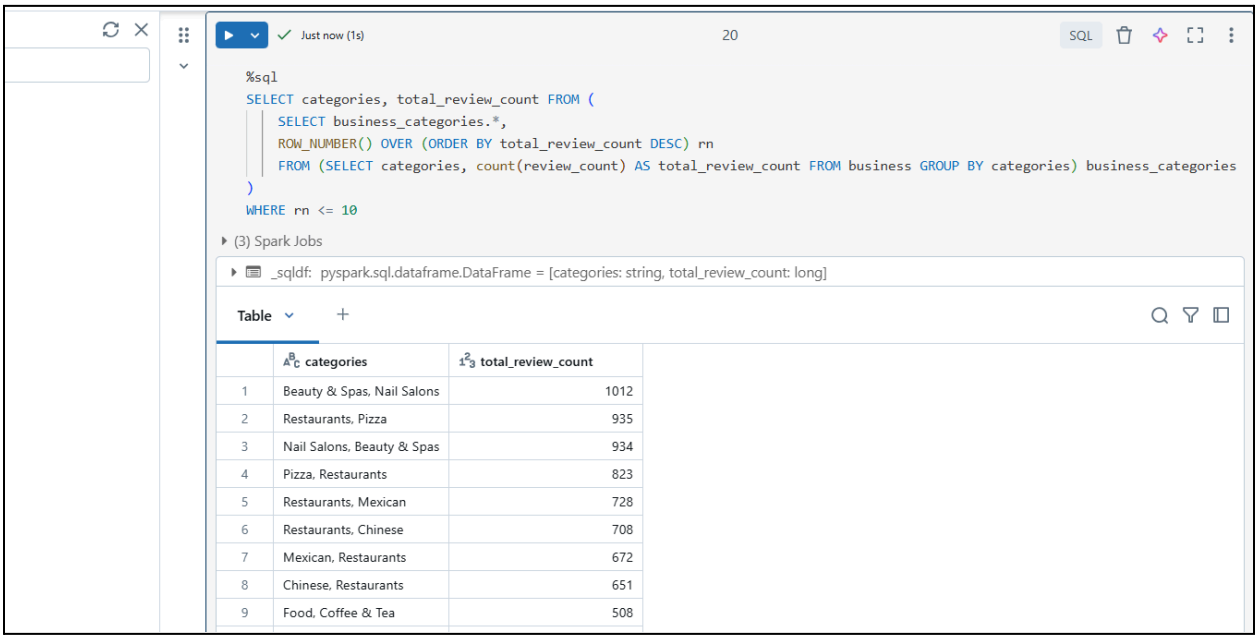
This code snippet enables SQL-based querying on the df_business DataFrame by creating a temporary view called "business". It makes the dataset accessible in subsequent cells using standard SQL queries for easier analysis and reporting.

21. Top 10 Categories by Total Review Count

```
SELECT categories, total_review_count FROM (
   SELECT business_categories.*,
   ROW_NUMBER() OVER (ORDER BY total_review_count DESC) rn
   FROM (
      SELECT categories, count(review_count) AS total_review_count
      FROM business
      GROUP BY categories
   ) business_categories
)
WHERE rn <= 10
```

```
SELECT categories, total_review_count FROM (
    SELECT business_categories.*,
    ROW_NUMBER() OVER (ORDER BY total_review_count DESC) rn
    FROM (
        SELECT categories, count(review_count) AS
```

```
 total_review_count
        FROM business
        GROUP BY categories
    ) business_categories
)
WHERE rn <= 10
```



	categories	total_review_count
1	Beauty & Spas, Nail Salons	1012
2	Restaurants, Pizza	935
3	Nail Salons, Beauty & Spas	934
4	Pizza, Restaurants	823
5	Restaurants, Mexican	728
6	Restaurants, Chinese	708
7	Mexican, Restaurants	672
8	Chinese, Restaurants	651
9	Food, Coffee & Tea	508

Explanation:

- This SQL query fetches the top 10 business categories based on the total number of reviews in descending order.

- The innermost subquery groups the `business` temporary view by the `categories` column and counts the number of `review_count` entries for each category.

- The outer subquery adds a row number to each grouped category using `ROW_NUMBER()` ordered by the `total_review_count` in descending order.

- The final `WHERE rn <= 10` clause filters the top 10 categories with the most review entries.

This SQL query returns the top 10 business categories that have received the highest number of review entries in the dataset. It achieves this by grouping the dataset by category, counting reviews, ranking them, and selecting the top 10.

22. Filtering Business Categories with High Review Counts

```
df_business_reviews =
df_business.groupBy("categories").agg(f.count("review_count").alias("total_review_co
unt"))
df_top_business =
df_business_reviews.filter(df_business_reviews["total_review_count"] >=
1000).orderBy(f.desc("total_review_count"))
```

```
df_business_reviews =
df_business.groupBy("categories").agg(f.count("review_count").al
ias("total_review_count"))
```

```
df_top_business =
df_business_reviews.filter(df_business_reviews["total_review_cou
nt"] >= 1000).orderBy(f.desc("total_review_count"))
```

Explanation:

- df_business.groupBy("categories"): This groups the dataset by the categories column.
- agg(f.count("review_count").alias("total_review_count")): The aggregation function counts the number of entries in the review_count column for each category, naming this new column total_review_count.
- df_business_reviews.filter(df_business_reviews["total_review_count"] >= 1000): Filters out any categories that have fewer than 1000 total reviews, keeping only those categories with 1000 or more reviews.
- orderBy(f.desc("total_review_count")): Orders the resulting categories in descending order by the total_review_count, so that categories with the highest review counts come first.

This code snippet processes the business data to identify categories with 1000 or more reviews. It groups the data by category, counts the reviews, filters categories based on review count, and sorts them in descending order to prioritize those with the most reviews.

23. This code will display the df_top_business DataFrame, which contains business categories with 1000 or more reviews, sorted in descending order by their total review count.

```
display(df_top_business)
```

Explanation:

- `display(df_top_business)`: This command will show the resulting DataFrame that includes the business categories with a total review count of 1000 or more, ordered by review count.

You should now be able to see the categories that are most reviewed in the Yelp dataset.

24. Number of Restaurants by State

```
df_num_of_restaurants =
df_business.select('state').groupBy('state').count().orderBy(f.desc("count"))
display(df_num_of_restaurants)
```

This code calculates and displays the number of restaurants per state, sorted by the state with the highest count of restaurants in descending order.

```
df_num_of_restaurants =
df_business.select('state').groupBy('state').count().orderBy(f.d
esc("count"))
display(df_num_of_restaurants)
```

Explanation:

- `df_business.select('state')`: This selects the `state` column from the `df_business` DataFrame, which represents the state of each business.
- `groupBy('state').count()`: Groups the data by state and counts the number of entries (restaurants) in each state.
- `orderBy(f.desc("count"))`: Orders the result in descending order based on the count, so the state with the most restaurants will be shown at the top.
- `display(df_num_of_restaurants)`: This will display the resulting DataFrame containing the number of restaurants for each state, sorted by the count in descending order.

By running this code, you will be able to see the states with the most restaurants in the dataset.

25. The code **`df_business.createOrReplaceTempView("business_restaurant")`**, creates or replaces a temporary view in Spark SQL, allowing you to run SQL queries against the `df_business` DataFrame.

Explanation:

- df_business.createOrReplaceTempView("business_restaurant"): This registers the df_business DataFrame as a temporary SQL view named business_restaurant. After this, you can write and run SQL queries using the business_restaurant table in Spark SQL. This view is only available during the current session and will be dropped once the session ends.

Once the temporary view is created, you can use SQL to query the data from the **df_business** DataFrame like so:

```
SELECT * FROM business_restaurant
```

This allows you to perform more complex analysis using SQL-style syntax in addition to using PySpark functions.

26. Top 3 Most Reviewed Restaurants by State

```
SELECT * FROM ( SELECT STATE,name,review_count, ROW_NUMBER() OVER (
PARTITION BY STATE ORDER BY review_count DESC) rn FROM business_restaurant )
WHERE rn <= 3
```

```
SELECT * FROM ( SELECT STATE,name,review_count, ROW_NUMBER()
OVER ( PARTITION BY STATE ORDER BY review_count DESC) rn FROM
business_restaurant ) WHERE rn <= 3
```

This SQL query retrieves the top 3 businesses in each state based on the review_count from the business_restaurant view created in Databricks.

Explanation:

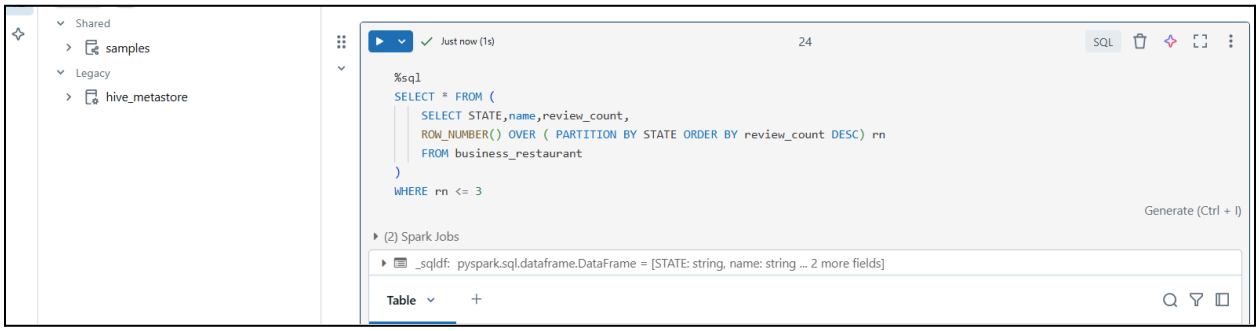
1. SELECT * FROM (...): This outer query selects all the rows from the inner query, which is used to filter and rank the businesses by their review count.

2. ROW_NUMBER() OVER (PARTITION BY STATE ORDER BY review_count DESC) rn: This part of the query assigns a row number (rn) to each business within each state (PARTITION BY STATE), ordered by the review_count in descending order. The highest review_count gets the row number 1, the second highest gets 2, and so on.

3. WHERE rn <= 3: This filters the result to only include the top 3 businesses for each state, as row numbers greater than 3 are excluded.

Example Result:

The output will be a list of businesses where each state will have up to three businesses ranked by their **review_count** in descending order.



This approach ensures that for each state, you get only the top 3 businesses based on their review count.

27. Top 10 Most Reviewed Restaurants in Arizona

```
df_business_Arizona = df_business.filter(df_business['state']=='AZ') df_Arizona = df_business_Arizona.groupBy("name").agg(f.count("review_count").alias("total_review_count")) window = Window.orderBy(df_Arizona['total_review_count'].desc()) df_Arizona_best_rest = df_Arizona.select('*', f.rank().over(window).alias('rank')).filter(f.col('rank') <= 10) display(df_Arizona_best_rest)
```

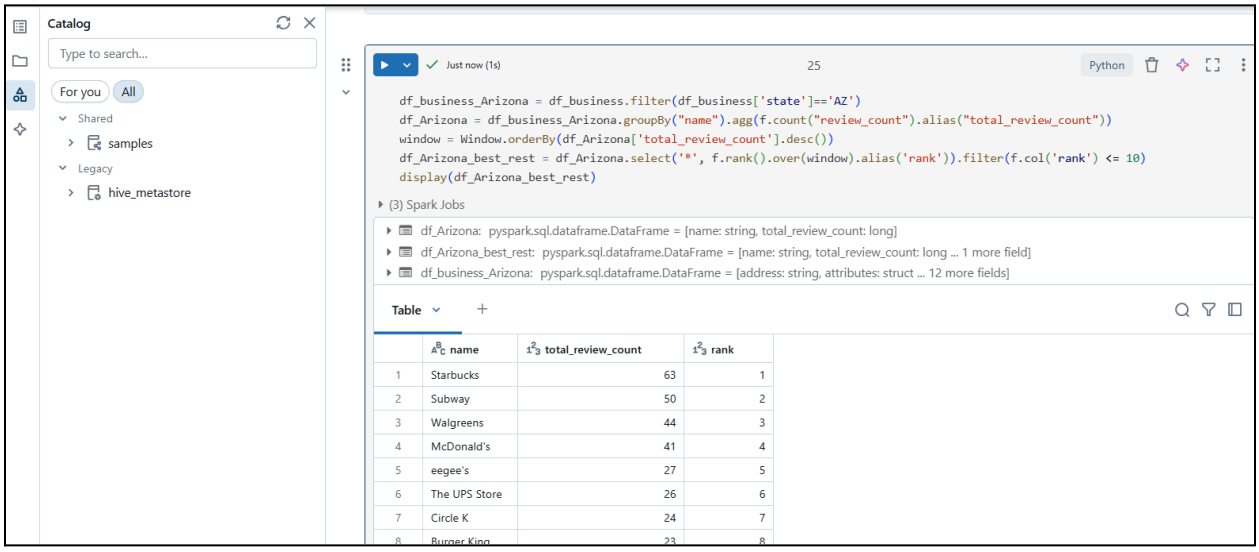

```
df_business_Arizona = df_business.filter(df_business['state']=='AZ') df_Arizona = df_business_Arizona.groupBy("name").agg(f.count("review_count").alias("total_review_count")) window = Window.orderBy(df_Arizona['total_review_count'].desc()) df_Arizona_best_rest = df_Arizona.select('*', f.rank().over(window).alias('rank')).filter(f.col('rank') <= 10) display(df_Arizona_best_rest)
```

This code focuses on analyzing Yelp business data for the state of Arizona (AZ). The goal is to filter and rank businesses based on their review count, and display the top 10 businesses in Arizona.

1. df_business_Arizona = df_business.filter(df_business['state'] == 'AZ'):
    - This line filters the original DataFrame df_business to include only businesses that are located in the state of Arizona ('AZ').
2. df_Arizona = df_business_Arizona.groupBy("name").agg(f.count("review_count").alias("total_review_count")):
    - After filtering for Arizona, the businesses are grouped by their name, and the total review count is aggregated using the count() function. This provides the total number of reviews for each business in Arizona.
3. window = Window.orderBy(df_Arizona['total_review_count'].desc()):
    - A window specification is created for ranking the businesses. The window orders the businesses by total_review_count in descending order, so businesses with the highest number of reviews appear first.
4. df_Arizona_best_rest = df_Arizona.select('*', f.rank().over(window).alias('rank')).filter(f.col('rank') <= 10):
    - The rank() function is used over the defined window to assign a rank to each business based on the total_review_count.
    - Then, the DataFrame is filtered to retain only the top 10 businesses (rank <= 10).

5. display(df_Arizona_best_rest):

    - Finally, the top 10 ranked businesses in Arizona are displayed in the Databricks environment.



This code filters businesses based in Arizona, calculates their total review count, and ranks them by this count in descending order. It then displays the top 10 businesses with the highest number of reviews.

28. Number of Restaurants in Each City in Arizona

```
df_business_Arizona = df_business.filter(df_business['state']=='AZ')
df_business_Arizona =
df_business_Arizona.groupBy('city').count().orderBy(f.desc("count"))
```


```
df_business_Arizona =
df_business.filter(df_business['state']=='AZ')
df_business_Arizona =
df_business_Arizona.groupBy('city').count().orderBy(f.desc("coun
t"))
```


Code Explanation:

This code snippet focuses on analyzing Yelp business data for the state of Arizona ('AZ'). Specifically, it groups the businesses by their city and counts the number of businesses in each city.

1. df_business_Arizona = df_business.filter(df_business['state'] == 'AZ'):
    - This line filters the df_business DataFrame to include only businesses located in Arizona (i.e., where the state is 'AZ').
2. df_business_Arizona = df_business_Arizona.groupBy('city').count().orderBy(f.desc("count")):
    - After filtering for businesses in Arizona, this line groups the data by the city column and counts how many businesses exist in each city.

- The orderBy(f.desc("count")) part orders the cities in descending order based on the number of businesses in each city, so that cities with the most businesses appear first.

This code filters Arizona-based businesses, then groups them by city and counts the number of businesses in each city. Finally, it orders the cities by the number of businesses in descending order, allowing us to see which cities have the most businesses in Arizona.

29. Visualise the number of restaurants in Arizona City, city Phoenix can be visualised from here, that is having maximum number of restaurants

display (df_business_Arizona)

This code is used to visualize the number of businesses (restaurants) in different cities within the state of Arizona, with a specific focus on Phoenix, the city having the maximum number of restaurants.

1. display(df_business_Arizona):

    - This command is used in Databricks to visualize the DataFrame df_business_Arizona in a tabular format.

    - It shows the count of businesses per city in Arizona, sorted in descending order of the count, allowing you to easily see which city has the most restaurants.

The display() function helps you view the df_business_Arizona DataFrame, which contains the count of restaurants in each city in Arizona. You can specifically look at Phoenix, the city with the highest number of restaurants, and analyze the data visually. This is an effective way to quickly gain insights into the distribution of restaurants across cities in Arizona.

30. Identify the City with the Maximum Number of Restaurants in Arizona

```
window = Window.orderBy(df_business_Arizona['count'].desc()) city_with_max_rest = df_business_Arizona \ .select('*', f.rank().over(window).alias('rank')).filter(f.col('rank') <= 1) \ .drop('rank') display(city_with_max_rest)
```

```
window = Window.orderBy(df_business_Arizona['count'].desc())
city_with_max_rest = df_business_Arizona \ .select('*',
f.rank().over(window).alias('rank')).filter(f.col('rank') <= 1)
\ .drop('rank') display(city_with_max_rest)
```

Code Explanation:

This code snippet is used to find the city with the maximum number of restaurants in Arizona. It ranks the cities based on the count of restaurants and then filters for the city with the highest rank (i.e., the one with the most restaurants).

1. Windowing and Ranking:

    - window = Window.orderBy(df_business_Arizona['count'].desc()):
      This defines a window that orders the cities in df_business_Arizona by the restaurant count in descending order.

    - f.rank().over(window).alias('rank'):
      The rank() function is applied over the defined window to assign a rank to each city based on the restaurant count, with the highest count receiving rank 1.

2. Filtering for Maximum Count:

    - .filter(f.col('rank') <= 1):
      This filters the DataFrame to keep only the city with the highest rank (i.e., the city with the maximum restaurant count).

    - .drop('rank'):
      Drops the 'rank' column since it's no longer needed for the output.

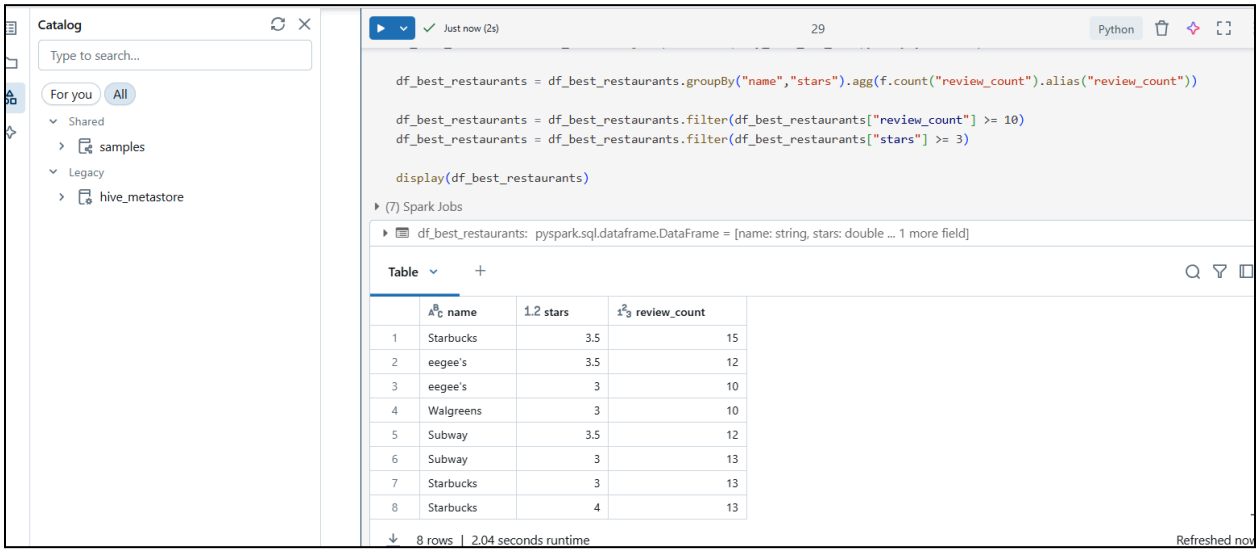
3. display(city_with_max_rest):
   This displays the final DataFrame, which contains the city with the highest number of restaurants.

The code identifies and displays the city in Arizona with the highest number of restaurants. It achieves this by ranking the cities based on their restaurant counts, filtering for the city with the top rank, and displaying the result. The visual output will clearly show the city with the most restaurants.

31. Best Restaurants in the City with the Most Restaurants in Arizona (e.g., Phoenix)

```
from pyspark.sql.functions import broadcast #Join arizona city dataset with business datasets to get more details df_best_restaurants = df_business.join(broadcast(city_with_max_rest),"city", 'inner') df_best_restaurants = df_best_restaurants.groupBy("name","stars").agg(f.count("review_count").alias("review_count")) df_best_restaurants = df_best_restaurants.filter(df_best_restaurants["review_count"] >= 10) df_best_restaurants = df_best_restaurants.filter(df_best_restaurants["stars"] >= 3)
```

```
from pyspark.sql.functions import broadcast #Join arizona city dataset with business datasets to get more details
df_best_restaurants = df_business.join(broadcast(city_with_max_rest),"city", 'inner')
df_best_restaurants = df_best_restaurants.groupBy("name","stars").agg(f.count("review_count").alias("review_count")) df_best_restaurants = df_best_restaurants.filter(df_best_restaurants["review_count"] >= 10) df_best_restaurants =
 df_best_restaurants.filter(df_best_restaurants["stars"] >= 3)
```



Code Explanation:

This code snippet joins the dataset of the top Arizona city with the business dataset to retrieve more detailed information about the restaurants, followed by filtering the results based on specific conditions such as the minimum review count and star rating.

1. Joining the Datasets:

    - df_best_restaurants = df_business.join(broadcast(city_with_max_rest), "city", 'inner'):
      The df_business DataFrame (which contains business details) is joined with city_with_max_rest (which contains the city with the most restaurants) using an inner join on the "city" column.
      The broadcast function is used to optimize the join by ensuring that the smaller DataFrame (city_with_max_rest) is broadcast to all worker nodes for better performance when joining.

2. Aggregating Data:

    - df_best_restaurants = df_best_restaurants.groupBy("name", "stars").agg(f.count("review_count").alias("review_count")):
      After the join, the dataset is grouped by "name" (the restaurant name) and "stars" (the restaurant's star rating). The count("review_count") is computed for each group to get the total number of reviews for each restaurant, and this count is aliased as "review_count".

3. Filtering the Results:

    - df_best_restaurants = df_best_restaurants.filter(df_best_restaurants["review_count"] >= 10):
      This filter ensures that only restaurants with at least 10 reviews are included in the dataset.
    - df_best_restaurants = df_best_restaurants.filter(df_best_restaurants["stars"] >= 3):
      This filter ensures that only restaurants with a star rating of 3 or higher are included in the dataset.

The code takes the city with the most restaurants from Arizona and joins it with the business dataset to get detailed restaurant information. It then groups the data by restaurant name and star rating while calculating the review count for each restaurant. Lastly, it filters the restaurants to include only those with at least 10 reviews and a rating of 3 stars or more.

32. Visualize restaurants as per review ratings in Phoenix city display(df_best_restaurants)

The **display(df_best_restaurants)** function will visualize the DataFrame df_best_restaurants, which contains the top-rated restaurants in Phoenix, based on their review counts and star ratings.

1. Visualizing the Data:

    - display(df_best_restaurants):
      This function call in Databricks will display the DataFrame df_best_restaurants. The DataFrame contains restaurant names, star ratings, and the review count, which were filtered in the previous steps.
      The table will show the restaurants in Phoenix city that meet the criteria (at least 10 reviews and a star rating of 3 or more), allowing you to visually inspect the data.

The displayed data will allow you to visualize the restaurants in Phoenix with their respective star ratings and review counts. This visualization helps to identify the best-rated restaurants in the city, offering insights into popular establishments based on user reviews and ratings.

33. Filtering Data for Phoenix and Italian Restaurants

The code filters the `df_business` DataFrame to focus on businesses located in Phoenix, and then narrows down the results to only include those with the category "Italian."

```
df_business_Pheonix = df_business.filter(df_business.city == 'Phoenix')
df_business_italian =
df_business_Pheonix.filter(df_business.categories.contains('Italian'))
```

```
df_business_Pheonix = df_business.filter(df_business.city ==
'Phoenix')
df_business_italian =
df_business_Pheonix.filter(df_business.categories.contains('Ital
ian'))
```

Explanation:

1. Filtering for Phoenix:

    - df_business_Pheonix = df_business.filter(df_business.city == 'Phoenix'):
      This line filters the df_business DataFrame to select only the rows where the city is "Phoenix." This operation reduces the dataset to just businesses located in Phoenix.

2. Filtering for Italian Restaurants:

    - df_business_italian = df_business_Pheonix.filter(df_business.categories.contains('Italian')):
      After filtering for Phoenix, this line further filters the DataFrame to include only those rows where the categories column contains the string "Italian." This operation narrows the dataset to include only Italian restaurants in Phoenix.

The code first selects all businesses located in Phoenix and then filters them further to include only those that fall under the "Italian" category. The resulting DataFrame df_business_italian contains a subset of the original data, focusing on Italian restaurants in Phoenix. This approach helps analyze a specific type of restaurant in a particular city.

34. Finding the Best Italian Restaurants in Phoenix

The following code groups the filtered dataset of Italian restaurants in Phoenix by restaurant name and counts the number of reviews for each. Then, it filters the results to include only those restaurants that have received at least 5 reviews.

```
df_best_italian_restaurants =
df_business_italian.groupBy("name").agg(f.count("review_count").alias("review_count
"))

df_best_italian_restaurants =
df_best_italian_restaurants.filter(df_best_italian_restaurants["review_count"] >= 5)
display(df_best_italian_restaurants)
```


```
df_best_italian_restaurants =
df_business_italian.groupBy("name").agg(f.count("review_count").
alias("review_count"))

df_best_italian_restaurants =
df_best_italian_restaurants.filter(df_best_italian_restaurants["
review_count"] >= 5)
display(df_best_italian_restaurants)
```

Explanation:

1. Grouping by Restaurant Name and Counting Reviews:

    - df_best_italian_restaurants = df_business_italian.groupBy("name").agg(f.count("review_count").alias("review_count")):
      This line groups the filtered Italian restaurants by their names and calculates the total number of reviews each restaurant has received. The result is stored in the new column review_count.

2. Filtering Restaurants with More than 5 Reviews:

    - df_best_italian_restaurants = df_best_italian_restaurants.filter(df_best_italian_restaurants["review_count"] >= 5):
      After counting the reviews, this line filters out restaurants that have fewer than 5 reviews, ensuring that only popular and well-reviewed restaurants are included in the final dataset.

This code snippet identifies and filters the best Italian restaurants in Phoenix by grouping the data by restaurant name and counting their reviews. It then filters out any restaurant with fewer than 5 reviews, ensuring that only the more established and reviewed Italian restaurants in Phoenix are selected. The resulting DataFrame, df_best_italian_restaurants, contains these filtered restaurants for further analysis or visualization.

Note:

The PySpark code snippets provided are structured to efficiently process and analyze Yelp business data stored in Azure Data Lake. The sequence begins with loading and counting rows from key datasets (tip, checkin, business, review, user), followed by **partitioning and optimizing storage** using coalesce and repartition. This prepares the data for faster queries and scalable processing.

The core analytical operations are designed to uncover **insights about user behavior and business performance**, such as:

- Identifying the **top 3 users by review count** and **top 10 users by number of fans**.
- Ranking **top business categories** based on review count using both SQL and DataFrame APIs.
- Determining the **number of restaurants per state** and the **top 3 most-reviewed restaurants in each state**.

The focus then narrows to **Arizona state**, and particularly the **city of Phoenix**, to discover:

- The city with the highest number of restaurants.
- The **best-rated restaurants in Phoenix** with over 10 reviews and at least 3 stars.
- A deeper dive into **Italian restaurants in Phoenix**, highlighting those with substantial review activity.

Each transformation and visualization step helps generate **location-specific business intelligence**, guiding where high-performing or popular businesses are concentrated.

You can also customize the PySpark code to process and analyze the data based on your specific needs and goals. To understand the logic of the code via video, you can also watch the project video series, starting from this **video onward (Parquet and Delta File Format).**

Summary

- In this project, we implemented an analytical ETL pipeline in Azure for processing and analyzing Yelp business data using PySpark and Azure cloud-native services.
- Azure Data Lake Gen2 served as the primary data lake, organizing raw, intermediate, and processed data layers in Parquet format for efficient storage and retrieval.
- A copy pipeline was created in **Azure Data Factory (ADF)** to automate the transfer of Yelp data from a source Azure Data Lake Storage Gen2 container to the destination storage container for further processing.
- We added a **trigger to the ADF pipeline**, ensuring that whenever a new data file lands in the storage container, the copy pipeline is automatically triggered, streamlining the data ingestion process.
- **Azure Databricks was used to run PySpark jobs**, performing data partitioning and optimization with repartition() and coalesce() to ensure faster queries and scalable processing.
- During transformation, the enriched data was written back to **Azure Data Lake**, enabling downstream analytics and reporting can also be done using tools like **Azure Synapse Analytics** and **Power BI**.
- The PySpark jobs cleaned, enriched, and standardized the data, handling missing values and optimizing the datasets **(conversion to parquet)** for efficient analytics.
    - Identified the top 3 users by review count and the top 10 users by fans.
    - Ranked top business categories by review count using both SQL and DataFrame APIs.
    - Analyzed the number of restaurants per state and the top 3 most-reviewed restaurants in each state.