SCS 3252 - FINAL PROJECT

Analyzing and Visualizing Seattle Micromobility Data

Dorothy Liu & Jessie Ma

Shareable electric scooters and bikes have risen in popularity over the past decade. They can be found in most major cities around the globe and especially in city centers where congestion makes it difficult for travelers to get around by car. As the demand for micromobility services increases so does the appearance of micromobility operators and with both - the need to exchange mobility data and improve operator services. In 2015, the North-American Bike-Share Association (NABSA) introduced the General Bikeshare Feed Specification (GBFS) to provide a standardized way for micromobility service providers to share publicly available real-time data feeds in order to increase visibility of mobility services. GBFS benefits everyone in the shared mobility space. For service operators, standardization allows for ease of implementing data requests. For consumers such as applications and software platforms, developers can efficiently aggregate data and compare feeds from multiple operators without customizing solutions. Finally, for cities and policymakers, GBFS enables them to easily ingest, transform, compare and analyze data feeds from all mobility operators in the city and make informed decisions to improve city-wide mobility plans.

In this study, we aim to uncover insights from GBFS micromobility data feeds in the city of Seattle in order to make informed decisions to enhance the city's mobility plan. We first scraped 12 hours of GBFS real-time data at one minute intervals from Lime, a micromobility provider in Seattle. We uploaded the files to DBFS to simulate a stream. We then processed and transformed the data in Databricks. Lastly, we visualized the data using two ways: using the built–in dashboards provided in Databricks and also using Power BI. To integrate with Power BI, we connected to Databricks Unity Catalog from Microsoft Power BI and developed a real-time micromobility dashboard. With a real-time operations dashboard, the City can potentially do the following and beyond:

1. Analyze demand in each district by time of day.
2. Instruct micromobility operators to reallocate assets to certain districts prior to the occurrence of peak demand.
3. Plan bike lanes and other supporting infrastructure in areas with high demand.
4. Identify underserved areas and develop strategies to address gaps and improve equity.
5. Make data-driven decisions on regulations, subsidies, and pilot programs.

Streaming and Visualizing in Databricks

HTTP Endpoint Streaming Problem

One of the deliverables for this project is a real-time dashboard of vehicle availability in Seattle by district. Lime has a public-facing API that provides bike status information when called, in GBFS format. Naturally, the initial approach was to see if the APIs can be read by spark.readStream to convert the json files into a Spark dataframe that we can then easily work with.

```
streamingInputDF = (
   spark.readStream
   .format("delta")
```

```
   .load(limeAPI))
```

However this approach soon proved to be unfruitful. We encountered the following error: `java.lang.UnsupportedOperationException`

This prompted an investigation into acceptable streaming sources for Spark. Spark documentation shows that some of the common input sources include Kafka and HDFS. Spark structured streaming does not support reading from HTTP endpoints. Further research shows that HTTP endpoints, such as the Lime APIs, generally do not make good streaming input sources for the following reasons:

1. Scalability: HTTP streaming implies a long-term open connection to the server, which is not scalable from the producer side. If streaming is allowed, a large number of open connections will consume a significant amount of resources, which can significantly increase overhead.
2. Nature of HTTP: HTTP's inherent design is request-response oriented, rather than for streaming. HTTP transactions are single requests from a client to a server, and corresponding responses from the server to the client.
3. Fault tolerance: HTTP protocol is stateless, whereas Spark streaming's fault tolerance requires the state of processing to be known. With a stateless protocol, it is difficult to keep track of the records being processed, and reprocess data if a failure does occur.
4. Rate limiting: Most HTTP endpoints have rate limiting imposed to prevent DDoS attacks. In a streaming context, this may impact the update frequency of the data source.

Given the above, it is evident that spark.readStream cannot be used for this project. The subsequent section explains the alternative used.

Streaming From File Source

Rather than streaming from the HTTP endpoint, we decided to pull the API data every minute (refer to Appendix A for the data scraping script used), save it to the DBFS, stream the saved json files, and do data analysis on those files. In order to have consistent testing files, we scraped 12 hours of Lime data every minute from the Lime API, and uploaded them to DBFS for the purpose of this project. In a real world scenario, we would simply scrape the data within databricks and save it to the DBFS, then call it from the DBFS with Spark streaming. The interval of one minute is chosen for 2 reasons:

1. It takes time to assign districts to bikes;
2. It is not necessary for the municipality to know exactly how many bikes are available in each district right at that second. One-minute is an acceptable refresh rate in most operating contexts.

Assigning Districts to Vehicles and Processing Geospatial Data in Databricks

GBFS data provides lat and lon of each mobility asset. It does not show which district/ neighbourhood the asset is currently located in, understandably so because GBFS is intended

to be a global standard for bikes and scooters, while each City has different districts. Districts thus have to be assigned to each record using a custom script.

Many cities have open data portals that provide data to the public for free, Seattle included. We retrieved Seattle's district data from their open data portal. The geojson file was downloaded and uploaded to DBFS for the district assignment script. Like the name suggests, geojson is a json-like file that is specifically designed for geographical information. Each district is a polygon expressed in a series of coordinates that define the boundaries of that district. The following script is used:

```
districtGDF = gpd.read_file("/seattle_district.geojson")
sparkDistrictGDF = sc.broadcast(districtGDF)
def assignDistrictToBike (lat,lon):
   mgdf = sparkDistrictGDF.value
   idx = mgdf[mgdf["geometry"].intersects(Point(lon, lat))].first_valid_index()
   return mgdf.loc[idx]["L_HOOD"] if idx is not None else None
findDistrictUDF = udf(assignDistrictToBike, StringType())
```

Note that the gpd (geopandas) library was used to create a geo dataframe. Geopandas is a popular python library that makes dealing with geospatial data easier. In databricks, there are three main ways of dealing with geospatial data at scale (reference: https://www.databricks.com/blog/2019/12/05/processing-geospatial-data-at-scale-with-databricks.html):

1. Using purpose-built Spark libraries for geospatial analytics such as GeoSpark, GeoMesa, GeoTrellis, and Rasterframes.
2. Wrapping single-node libraries such as GeoPandas in ad-hoc user defined functions (UDFs) for processing in a distributed fashion with Spark DataFrames.
3. Indexing the data with grid systems and leveraging the generated index to perform spatial operations.

Evidently approach #2 was used. For one, the project team is very familiar with the geopandas library as we have used it before in other projects for geospatial analysis. Secondly, we tested the speed of this script and found the processing time to be acceptable for what we are building. The broadcast function enables parallel processing which helps speed up the script. During the development of this script, we ran into a very specific problem with DBFS, as explained below.

The Lime Schema and Streaming DF

The Lime schema follows the standard GBFS format:

```
limeSchema = StructType([
   StructField("last_updated", IntegerType(), False),
   StructField("ttl", IntegerType(), False),
   StructField("version", StringType(), False),
   StructField("data", StructType([
```

```
        StructField("bikes", ArrayType(StructType([
            StructField("bike_id", StringType(), False),
            StructField("lat", DoubleType(), False),
            StructField("lon", DoubleType(), False),
            StructField("is_reserved", IntegerType(), False),
            StructField("is_disabled", IntegerType(), False),
            StructField("vehicle_type", StringType(), False)
        ])), False)]))])
```

We are not concerned with version and ttl in this assignment. Rather, we are interested in each individual bike record, as well as last_updated. The last_updated value was casted to every single bike record within a file for timestamping purpose, using the following code:

```
streamingInputDF = (spark
   .readStream
   .schema(limeSchema)
   .option("maxFilesPerTrigger", 1)
   .json(inputPath))
explodedDF = streamingInputDF.select(
   explode("data.bikes").alias("bikes"),
   col("last_updated").cast("timestamp"))
bikesWithDistrictDF = explodedDF.withColumn(
   "district",findDistrictUDF(col("bikes.lat"), col("bikes.lon"))
   ).select("bikes.*", "district","last_updated")
```

The explode function extracts each bike record as its own row, which retains only relevant information within the resulting dataframe, and makes downstream processing much easier. The last_updated column from the original streamingInputDF is casted to every record. Finally, the UDF defined earlier is used to find the district each bike is located in, and only relevant columns are selected for eventual querying.

```
streamingCountsDF = (
 bikesWithDistrictDF
   .groupBy(
     bikesWithDistrictDF.district,
     window(bikesWithDistrictDF.last_updated, "1 second"))
   .count()
streamingCountsDF.createOrReplaceTempView("bikeCountPerDistrict")
```

We are not applying a watermark here since we are not expecting data to arrive late. In real operations, the data is fetched once every minute from the HTTP endpoint, but it is arriving as one file, so a watermark should not be needed either. Note that the window is set to be 1 second, since every single record within the same file is all fetched at once and is thus guaranteed to have the same timestamp (i.e. no late arriving data).

The Query Itself

The query itself was fairly simple:

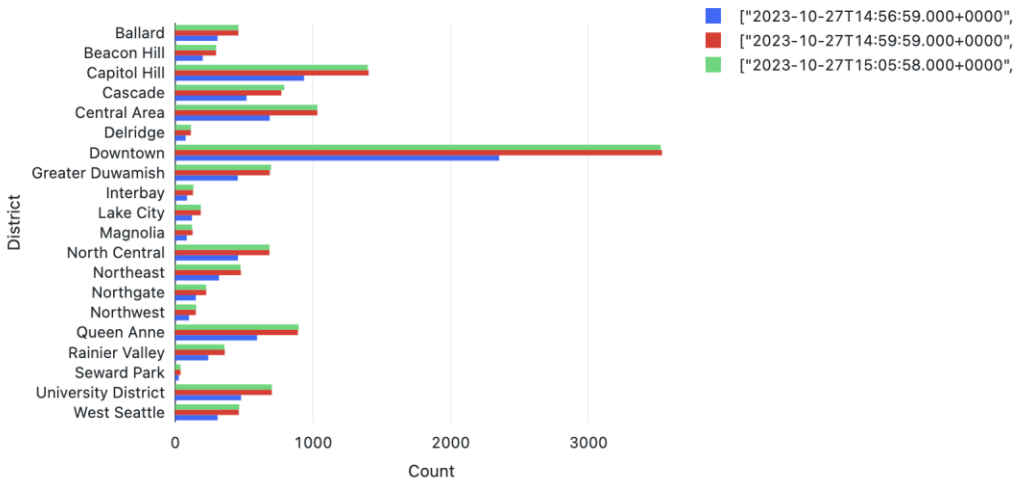
```
query = (streamingCountsDF
   .writeStream
   .outputMode("complete")
   .format("memory")
   .queryName("bikeCount")
   .start())
```

We didn't put a trigger condition in the query since it is streaming from a static file source. In a real world scenario where new data is being pulled every minute, we can place a trigger of 1 minute to save on some resources - rather than continuously checking for new data, Spark only checks for new data every minute.

The Graphs

The graphs were generated using Databricks' built-in visualization tool. When the display function is called, databricks allows users to set up customized visualization views in a simple way.

```
display(spark.table("bikeCountPerDistrict"))
```

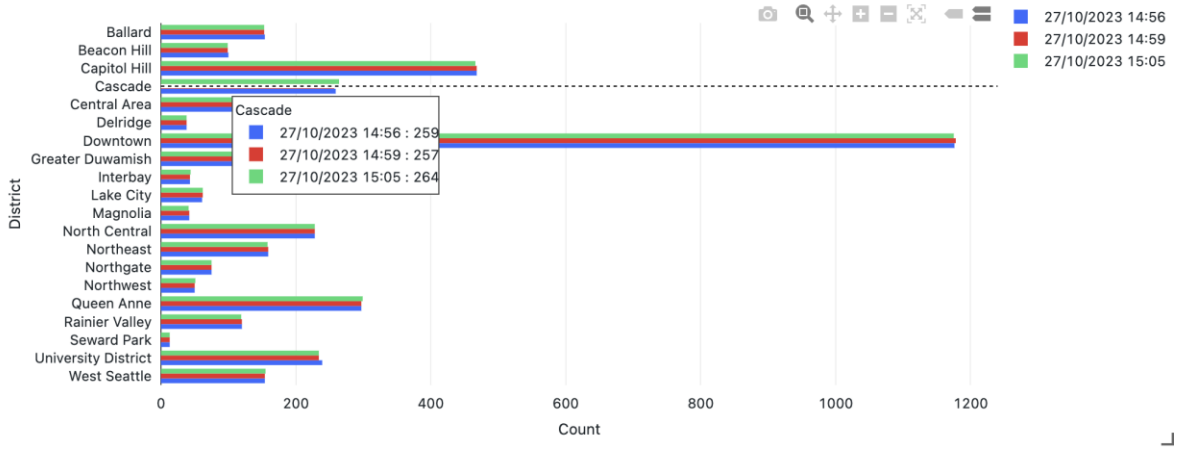


The last big issue we encountered was that the count was doubling at times, which suggests something wrong with the source data. Going back to the json files, it became evident that the last_updated value only changes if there is an actual change in the number of available bikes, so if all bike statuses remain the same across all Lime assets, the last_update value stays the same even if the data was fetched a minute apart. Another tell was that in the legend, the beginning of each window was not a minute apart. This explains why some counts are doubled. We added the following line to drop files with duplicate last_updated values.

```
dropDuplicatedDF = streamingInputDF.dropDuplicates(['last_updated'])
```

As shown below, this resolved our issue of double/triple counting. We also ended up using a SQL query for visualization to select window.start to make the legend more readable.

```
SELECT district, `count`, window.start FROM bikeCountPerDistrict
```



Power BI Connection and Visualization

Data Cleansing Prior to Ingestion

Similar to what was done using the streaming method the JSON files were first cleansed (refer to Appendix B for the Python script).

The cleansed JSON file looks like so:

```
{
   "bike_id": "c80f6da3-d152-4477-a7d1-4b377420828c",
    "lat": 47.6073,
    "lon": -122.3328,
    "is_reserved": false,
    "is_disabled": false,
    "vehicle_type": "scooter",
    "last_updated": 1698419158
}
```

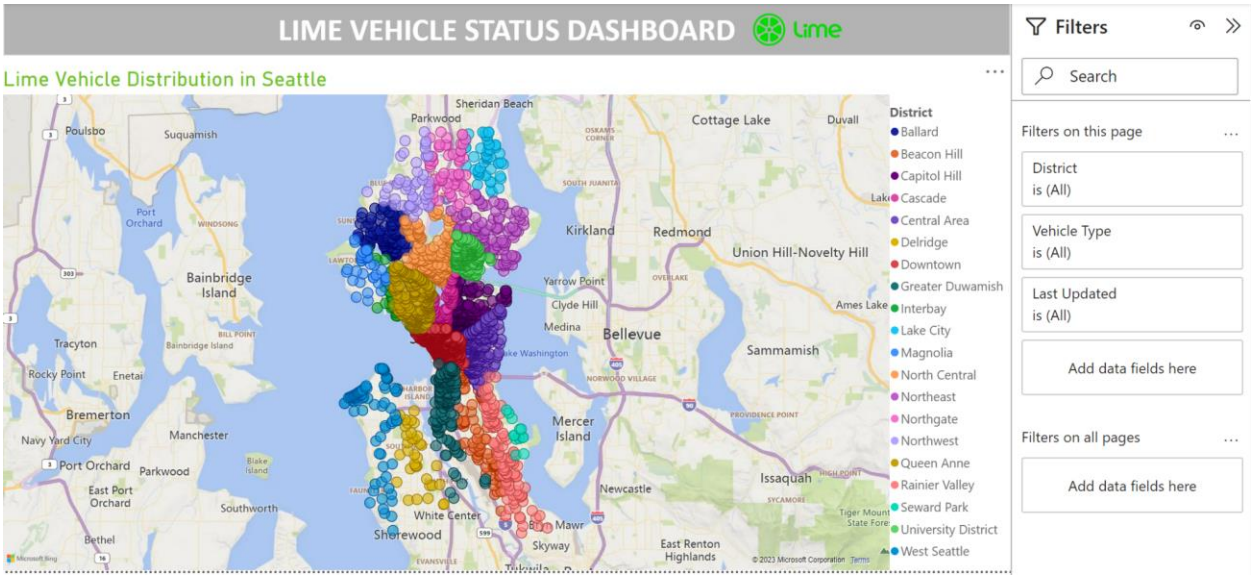
Since we would like to analyze a snapshot of the bike status at a point in time, we uploaded one JSON file (one timestamp) to be read into a Pyspark dataframe for further processing. Refer to Appendix C for the published Databricks notebook of the full list of commands discussed in this section, including the transformations and processing performed on the Lime dataframe.

One advantage of using databricks is its ability to be connected to external tools, whether they be databases, streaming platforms, visualization tools, or GitHub. To explore the integration ability of databricks, we decided to connect the dataframe to Power BI to provide more comprehensive visualization. We achieved this by writing the data frame into a delta table, then connecting the Power BI desktop version to Databricks to create two dashboards.

Here are some of the main insights we uncovered from processing the file:

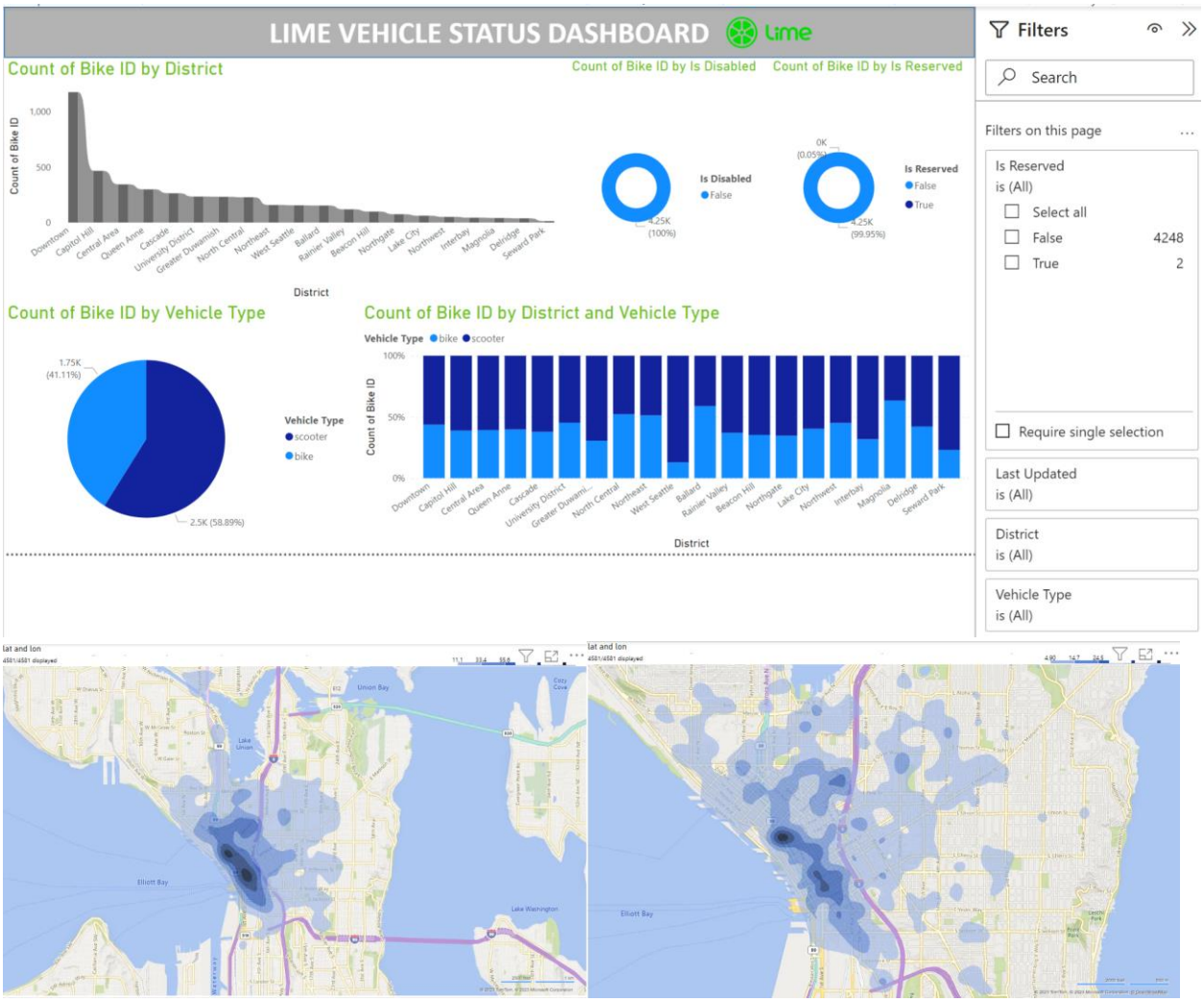
a. As noted from the Databricks graphs created via streaming in the first section, we can confirm from queries that Downtown, Capitol Hill and Central Area are the top 3 districts with the highest number of Lime vehicles. It is interesting to note that Downtown has more than twice the number of bikes available in the district with the second highest count of bikes. This makes sense as the Downtown district tends to be more populated than other districts; but are all of these available bikes actually being occupied? As policy analysts we may need to dig further when we can visualize this dataset in Power BI.
b. There are more scooters than bikes available from the data feed.
c. Downtown, Capitol Hill and Central Area had the highest count of scooters and bikes compared to other districts.

The first dashboard visualizes geographically the dispersion of Lime vehicles across the city. Users can filter on specific districts, vehicle types and last updated timestamps.



The second dashboard consists of a few different visuals for drilled-down details on the counts and distribution of vehicle types by districts. One noteworthy find is the high number of non-reserved Lime vehicles (4248) available in the city. On the one hand, it makes sense that since most mobility assets are located downtown, the number of free bikes will be higher. However, without knowing the number of bikes in use (this is not provided by public GBFS data due to privacy concerns), it is impossible to tell what's the percentage of available bikes compared to unavailable bikes for each district. If the City has data on bikes in-use in addition to available bikes, more comprehensive data analysis can be conducted to gauge the overall supply and

demand of each district, and determine whether some districts are oversupplied, and some may need more devices to meet demand.



Summary

To conclude in this report we demonstrated the full pipeline of ingesting, processing and displaying GBFS lime data. We ingested the data via two methods, by streaming and standard dataframe reading, cleansed the data, performed Pyspark queries and visualized the data in Databricks and Power BI. The conclusions we drew on the data merely scratch the surface of the potential this data unlocks. Policymakers and other stakeholders may continue to use these dashboards to further understand and analyze the micromobility demands of Seattle.

Appendices

A - Python script to scrape live Lime data feed

```
import requests
import time
from datetime import datetime, time

api_url_lime =
"https://data.lime.bike/api/partners/v1/gbfs/seattle/free_bike_status.json"
output_directory_lime = "lime_data/"

def fetch_and_save_data():
   response_lime = requests.get(api_url_lime)
   timestamp = int(time.time())
   filename_lime = f"{output_directory_lime}{timestamp}_lime.json"

   with open(filename_lime, 'wb') as file:
           file.write(response_lime.content)

if __name__ == "__main__":
   while True:
       fetch_and_save_data()
       time.sleep(59)
```

B - Python Script to modify Lime JSON files

In [8]:

```python
import os
import json

def update_json_file(input_file, output_path):
    with open(input_file, 'r') as file:
        data = json.load(file)
        last_updated = data.get('last_updated')
        new_data = data.get('data', {}).get('bikes')
        new_data2 = [dict(row, last_updated=last_updated) for row in new_data]

    for entry in new_data2:
        if "is_reserved" in entry and entry["is_reserved"] == 0:
            entry["is_reserved"] = False
        elif "is_reserved" in entry and entry["is_reserved"] == 1:
            entry["is_reserved"] = True

        if "is_disabled" in entry and entry["is_disabled"] == 0:
            entry["is_disabled"] = False
        elif "is_disabled" in entry and entry["is_disabled"] == 1:
            entry["is_disabled"] = True

       output_folder = os.path.join(output_path, 'lime_lastupdate')
    os.makedirs(output_folder, exist_ok=True)

    with open(os.path.join(output_folder, os.path.basename(input_file)), 'w') as file:
        json.dump(new_data2, file, indent=2)

def process_files_in_directory(directory):
    for filename in os.listdir(directory):
        if filename.endswith('.json'):
            input_file = os.path.join(directory, filename)
            update_json_file(input_file, directory)

if __name__ == "__main__":
       directory_path = r'C:\UofT SCS - 3252\Final Project\lime'

    process_files_in_directory(directory_path)
```

C - Published Databricks notebooks

Published Databricks Notebooks:
Streaming and built in visualizations in Databricks - https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/7701764096770436/2678307937317778/4109468684936282/latest.html

Standard ingestion and Power BI integration from Databricks - https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/7701764096770436/903100961516351/4109468684936282/latest.html

D - Power BI File

https://drive.google.com/file/d/1Amxqz81WOpSvH4goWdZSHwTuynl8neRP/view?usp=drive_link