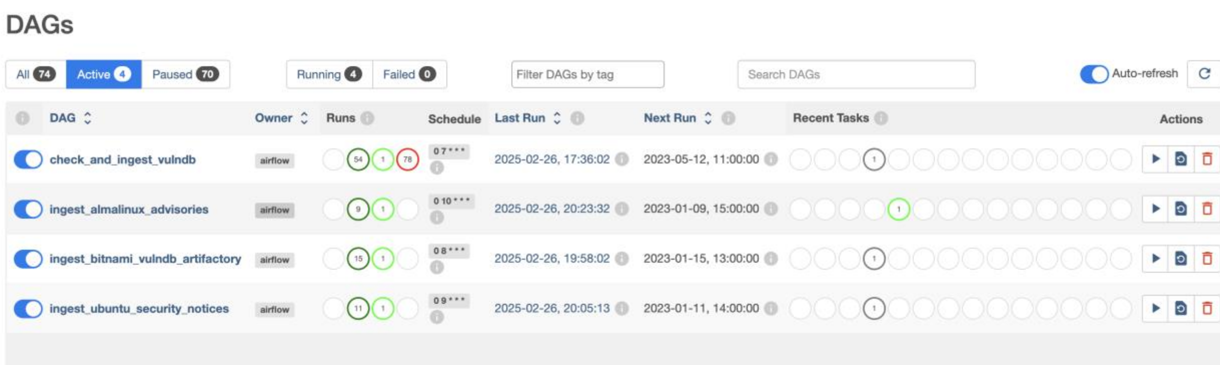
Data Ingestion

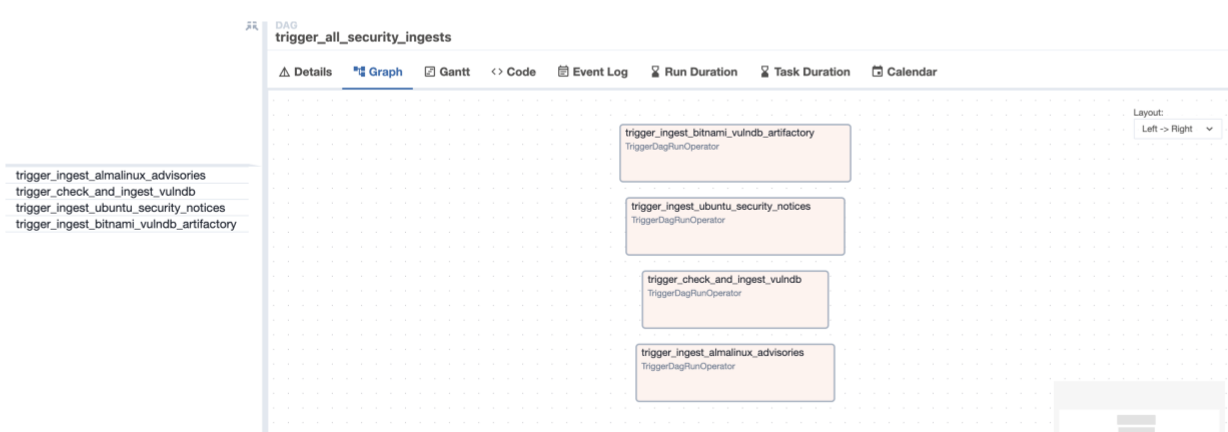
1. Daily Data Ingestion from GitHub to Azure Blob Storage

- **Scheduled Daily Execution:**
  Airflow DAGs are set up to run at distinct times (for example, 7 AM, 8 AM, 9 AM, and 10 AM ET) so that data from different OSV current data sources is fetched once every day. This scheduling ensures that the system automatically retrieves the latest updates without manual intervention.
- **Comprehensive File Retrieval:**
  A recursive file discovery function is implemented to traverse GitHub's API directories. This function checks each item to confirm it's a file (and not a directory) and that its extension is JSON. This approach guarantees that all JSON files—even those nested within subdirectories—are identified and ingested every day.
- **Schema Validation and Data Quality Checks:**
  During ingestion, key fields such as `id`, `aliases`, `affected`, and `modified` are implicitly validated. Files that don't meet these criteria are logged and skipped. This check maintains data quality by ensuring that only records with the necessary information proceed to the next stage.
- **Robust Error Handling and Retries:**
  Every API call checks the HTTP response code. If a file fails to download (for example, due to a non-200 status), the error is logged and that file is skipped. Airflow's built-in retry mechanism further helps to address transient errors, ensuring that the daily ingestion process remains resilient.
- **Audit Logging:**
  The system logs the names of successfully ingested files in Airflow Variables. This creates an audit trail for each daily run, making it easier to trace back and troubleshoot any issues that may arise later.
- **Uploading to Azure Blob Storage:**
  After validation, JSON files are temporarily stored locally (typically in a `/tmp` directory) before being uploaded to Azure Blob Storage using Airflow's WasbHook. Files are stored in designated containers (such as `vulalmalinux`, `vulbitnami`, `vulubunto`, or `vulgo`), ensuring that the data is well-organized by source.

.

Pipeline Screenshots

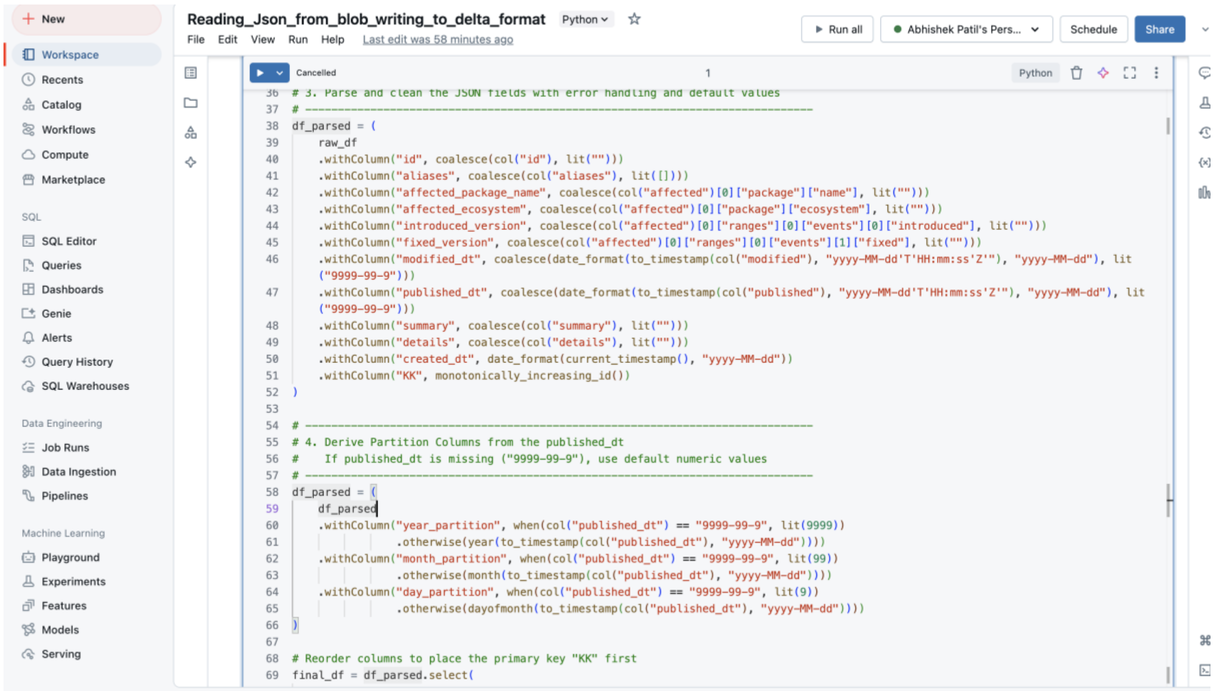


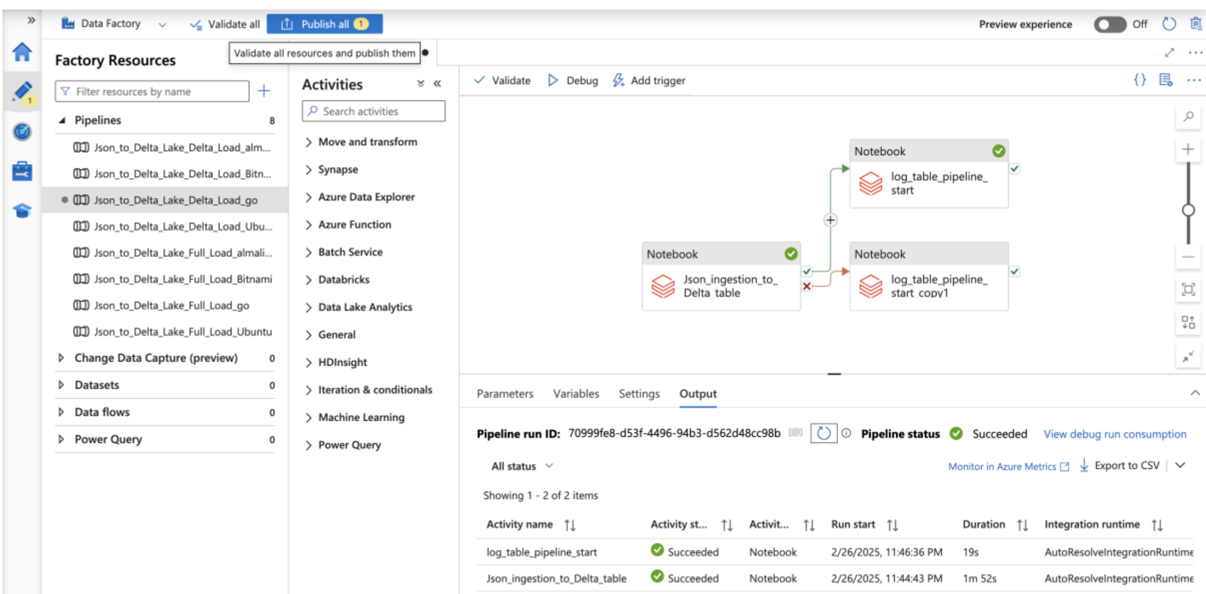


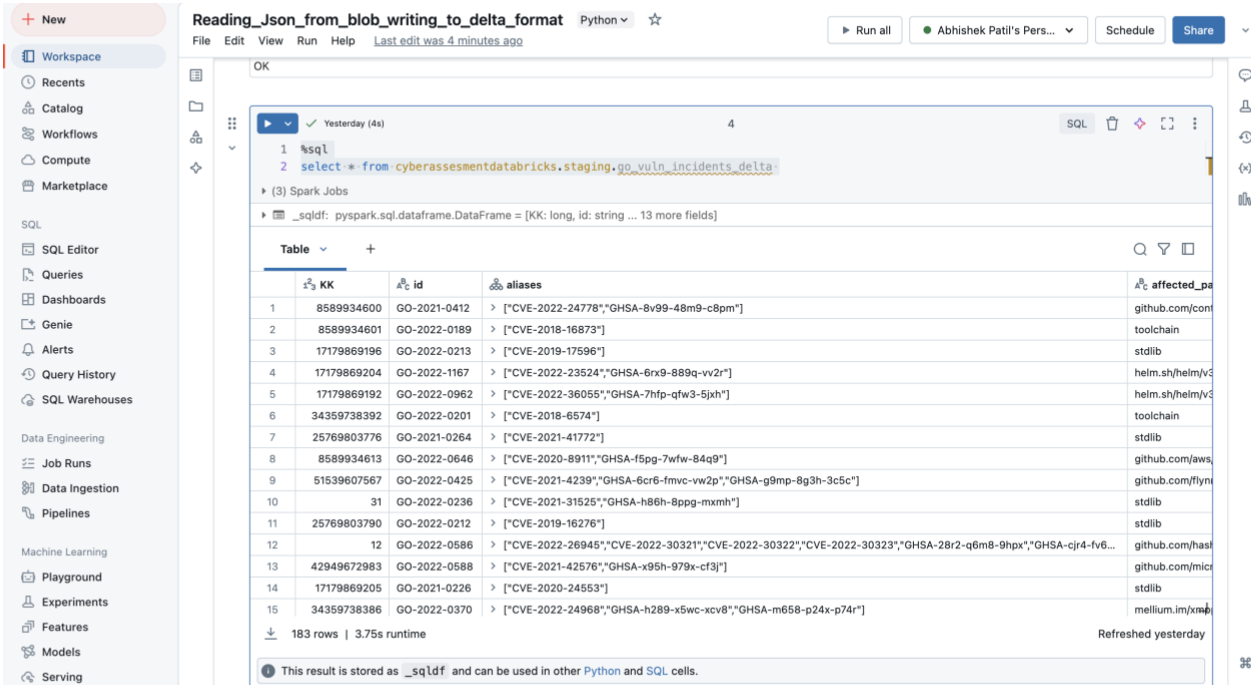
2. Data Ingestion from Azure Blob Storage to Delta Lake

1. **Reading from Azure Blob Storage**
    - A Databricks notebook is triggered daily via Azure Data Factory to process the raw JSON files residing in Azure Blob Storage. Spark's JSON reader is configured with the `multiline` option, enabling it to handle JSON records that may span multiple lines.
    - This setup ensures that all JSON data, validated during the Airflow ingestion phase, is loaded accurately and consistently into the Spark environment.
2. **Data Cleaning and Enrichment**
    - **Normalization:** Functions like `coalesce` replace any null values with sensible defaults, guaranteeing a consistent schema across all records.
    - **Nested Field Extraction:** Key attributes (e.g., `affected_package_name` and version details) are extracted from deep within the JSON structure. This step ensures that all relevant data points are readily accessible for downstream processing and queries.
    - **Timestamp Standardization:** Fields representing time are converted using `to_timestamp` and `date_format`, providing a uniform date-time format that simplifies both partitioning and time-based analytics.
    - **Audit Enhancements:** Each record receives a creation timestamp and a unique identifier generated by `monotonically_increasing_id()`. This approach facilitates traceability for daily loads and aids in debugging or rollback scenarios.
3. **Efficient Partitioning Strategy**
    - Partition columns—such as `year_partition`, `month_partition`, and `day_partition`—are derived from a standardized `published_dt` field. If a record's date is invalid or missing, default values (like 9999, 99, and 9) are applied, ensuring that all data is properly partitioned without errors.
    - This time-based partitioning allows for efficient queries, especially when filtering by specific date ranges or analyzing historical trends.
4. **Idempotent Writes to Delta Lake**
    - The final enriched DataFrame is written to Delta Lake in overwrite mode, guaranteeing that duplicate records are not introduced if the job re-runs on the same day.
    - Delta Lake's ACID properties enable reliable transaction management, while time travel features allow historical snapshots to be queried and rolled back if needed.

Databricks Notebook and Azure Data Factory Pipelines






New
Workspace
Recents
Catalog
Workflows
Compute
Marketplace
SQL
SQL Editor
Queries
Dashboards
Genie
Alerts
Query History
SQL Warehouses
Data Engineering
Job Runs
Data Ingestion
Pipelines
Machine Learning
Playground
Experiments
Features
Models
Serving
Reading_Json_from_blob_writing_to_delta_format
Python
File Edit View Run Help Last edit was 4 minutes ago
Run all
Abhishek Patil's Pers...
Schedule
Share
OK
Yesterday (4s)
4
SQL
1 %sql
2 select * from cyberassesmentdatabricks.staging.go_vuln_incidents_delta
(3) Spark Jobs
_sqldf: pyspark.sql.dataframe.DataFrame = [KK: long, id: string ... 13 more fields]
Table
KK id aliases affected_pa
1 8589934600 GO-2021-0412 ["CVE-2022-24778","GHSA-8v99-48m9-c8pm"] github.com/con
2 8589934601 GO-2022-0189 ["CVE-2018-16873"] toolchain
3 17179869196 GO-2022-0213 ["CVE-2019-17596"] stdlib
4 17179869204 GO-2022-1167 ["CVE-2022-23524","GHSA-6rx9-889q-vv2r"] helm.sh/helm/v3
5 17179869192 GO-2022-0962 ["CVE-2022-36055","GHSA-7hfp-qfw3-5jxh"] helm.sh/helm/v3
6 34359738392 GO-2022-0201 ["CVE-2018-6574"] toolchain
7 25769803776 GO-2021-0264 ["CVE-2021-41772"] stdlib
8 8589934613 GO-2022-0646 ["CVE-2020-8911","GHSA-f5pg-7wfw-84q9"] github.com/aws
9 51539607567 GO-2022-0425 ["CVE-2021-4239","GHSA-6cr6-fmvc-vw2p","GHSA-g9mp-8g3h-3c5c"] github.com/flyn
10 31 GO-2022-0236 ["CVE-2021-31525","GHSA-h86h-8ppg-mxmh"] stdlib
11 25769803790 GO-2022-0212 ["CVE-2019-16276"] stdlib
12 12 GO-2022-0586 ["CVE-2022-26945","CVE-2022-30321","CVE-2022-30322","CVE-2022-30323","GHSA-28r2-q6m8-9hpx","GHSA-cjr4-fv6... github.com/has
13 42949672983 GO-2022-0588 ["CVE-2021-42576","GHSA-x95h-979x-cf3j"] github.com/mic
14 17179869205 GO-2021-0226 ["CVE-2020-24553"] stdlib
15 34359738386 GO-2022-0370 ["CVE-2022-24968","GHSA-h289-x5wc-xcv8","GHSA-m658-p24x-p74r"] mellium.im/xm
183 rows | 3.75s runtime
Refreshed yesterday
This result is stored as _sqldf and can be used in other Python and SQL cells.

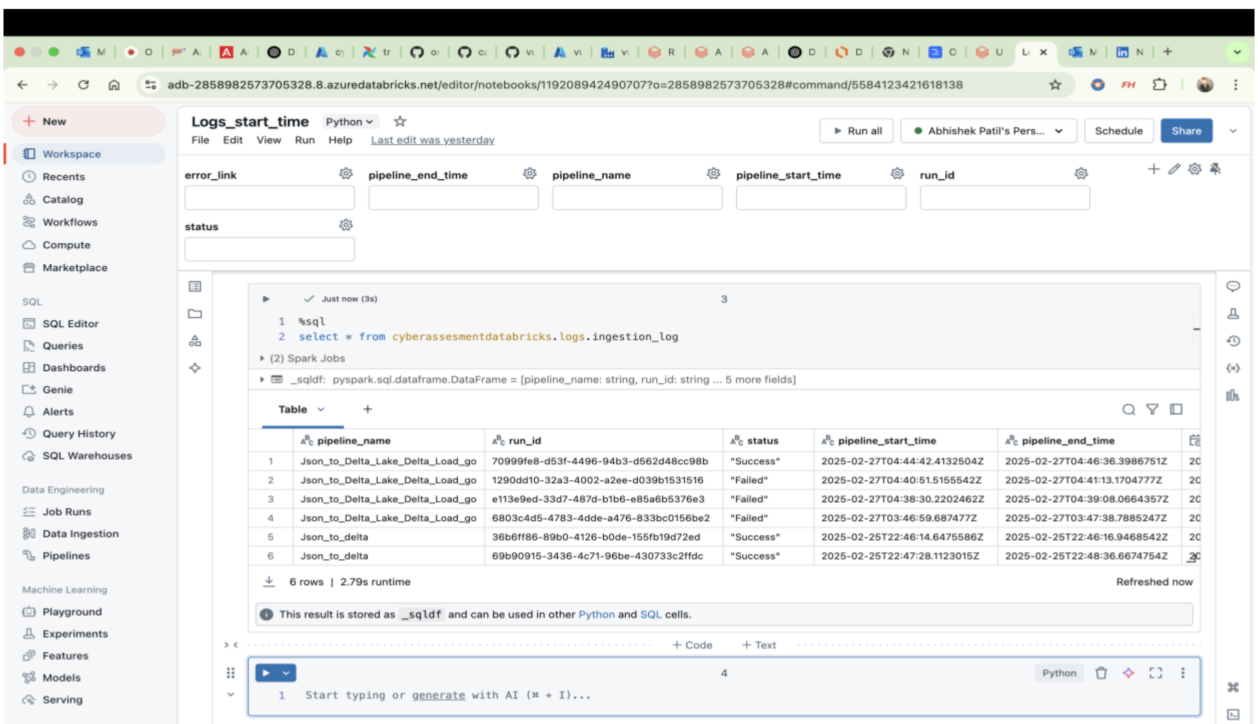
Code -

```
%python

from pyspark.sql.functions import (
    col,
    year,
    month,
    dayofmonth,
    to_timestamp,
    current_timestamp,
    coalesce,
    lit,
    date_format,
    when,
    monotonically_increasing_id
)

# -----------------------------------------------------------------------------
# 1. Set Azure Storage Configurations
# -----------------------------------------------------------------------------
spark.conf.set("fs.azure.account.key.cybersecurityproject.dfs.core.windows.net",
               "Rjy2JCqN7HaKdK+KR7Qkr1Gug2/CiqGh4LDVfZo+yS27rrAkuKNJucq2tXgE1a6ddjTMzDHiaZn5+AStNm2JUA==")
spark.conf.set("fs.azure.account.key.cyberdatalake1.dfs.core.windows.net",
               "dy/dqb6i330erjxGhssVcj9pm0nfy+nTxPsprOV5EmEEiJf3A+9+HyyHzXhLE315ypwvkXTXu/vD+AStj3Hs7A==")

# -----------------------------------------------------------------------------
# 2. Read the multi-line JSON from the source container
#    - Optionally, set a badRecordsPath to capture the rejected records.
# -----------------------------------------------------------------------------
raw_df = (
    spark.read
    .option("multiline", "true")
    .option("badRecordsPath", "abfss://deltalakestorage1@cyberdatalake1.dfs.core.windows.net/delta/badRecords")
    .json("abfs://vulgo@cybersecurityproject.dfs.core.windows.net/")
)

# -----------------------------------------------------------------------------
# 3. Parse and clean the JSON fields with error handling and default values
# -----------------------------------------------------------------------------
df_parsed = (
    raw_df
    .withColumn("id", coalesce(col("id"), lit("")))
    .withColumn("aliases", coalesce(col("aliases"), lit([])))
```

```python
    .withColumn("affected_package_name", coalesce(col("affected")[0]["package"]["name"], lit("")))
    .withColumn("affected_ecosystem", coalesce(col("affected")[0]["package"]["ecosystem"], lit("")))
    .withColumn("introduced_version", coalesce(col("affected")[0]["ranges"][0]["events"][0]["introduced"], lit("")))
    .withColumn("fixed_version", coalesce(col("affected")[0]["ranges"][0]["events"][1]["fixed"], lit("")))
    .withColumn("modified_dt", coalesce(date_format(to_timestamp(col("modified"), "yyyy-MM-dd'T'HH:mm:ss'Z'"), "yyyy-MM-dd"), lit("9999-99-9")))
    .withColumn("published_dt", coalesce(date_format(to_timestamp(col("published"), "yyyy-MM-dd'T'HH:mm:ss'Z'"), "yyyy-MM-dd"), lit("9999-99-9")))
    .withColumn("summary", coalesce(col("summary"), lit("")))
    .withColumn("details", coalesce(col("details"), lit("")))
    .withColumn("created_dt", date_format(current_timestamp(), "yyyy-MM-dd"))
    .withColumn("KK", monotonically_increasing_id())
)

# -----------------------------------------------------------------------------
# 4. Derive Partition Columns from the published_dt
#    If published_dt is missing ("9999-99-9"), use default numeric values
# -----------------------------------------------------------------------------
df_parsed = (
    df_parsed
    .withColumn("year_partition", when(col("published_dt") == "9999-99-9", lit(9999))
                .otherwise(year(to_timestamp(col("published_dt"), "yyyy-MM-dd"))))
    .withColumn("month_partition", when(col("published_dt") == "9999-99-9", lit(99))
                .otherwise(month(to_timestamp(col("published_dt"), "yyyy-MM-dd"))))
    .withColumn("day_partition", when(col("published_dt") == "9999-99-9", lit(9))
                .otherwise(dayofmonth(to_timestamp(col("published_dt"), "yyyy-MM-dd"))))
)

# Reorder columns to place the primary key "KK" first
final_df = df_parsed.select(
    "KK",
    "id",
    "aliases",
    "affected_package_name",
    "affected_ecosystem",
    "introduced_version",
    "fixed_version",
    "modified_dt",
    "published_dt",
    "summary",
    "details",
    "created_dt",
    "year_partition",
    "month_partition",
    "day_partition"
```

```
)

# Display schema and a sample of the data for verification
final_df.printSchema()
display(final_df)

# ---------------------------------------------------------------------------
# 5. Write the cleaned DataFrame to a Delta table with partitioning
# ---------------------------------------------------------------------------
final_df.write \
    .format("delta") \
    .mode("overwrite") \
    .partitionBy("year_partition", "month_partition", "day_partition") \
    .save("abfss://deltalakestorage1@cyberdatalake1.dfs.core.windows.net/delta/output")
```

Log Table



Create Table Statement

I built tables on top of parquet file locations for efficient querying

```
CREATE SCHEMA IF NOT EXISTS cyberassesmentdatabricks.curated;

CREATE TABLE IF NOT EXISTS cyberassesmentdatabricks.curated.go_vuln_incidents_delta (
   id STRING,
   aliases ARRAY<STRING>,
   affected_package_name STRING,
   affected_ecosystem STRING,
   introduced_version STRING,
   fixed_version STRING,
   modified_dt TIMESTAMP,
   published_dt TIMESTAMP,
   summary STRING,
   details STRING,
   created_dt TIMESTAMP,
   year_partition INT,
   month_partition INT,
   day_partition INT
)
USING DELTA
PARTITIONED BY (year_partition, month_partition, day_partition)
LOCATION 'abfss://deltalakestorage1@cyberdatalake1.dfs.core.windows.net/delta/output';
```