Databricks: Dynamically Generating Tables with DLT

Ryan Chynoweth

5 min read · Sep 19, 2022

Introduction

Prior to joining Databricks I worked as a cloud consultant, mostly working in Azure but with some AWS experience as well. I first started using Apache Spark in 2015, and when I discovered Databricks sometime in 2017 I was truly pleased with how easy and powerful the product was. Then when Azure Databricks was released I was thrilled to be able to start using even more since most of my projects were Azure based. At first, Delta Lake was not generally available and we had to work with mostly with parquet tables, however, when Delta Lake was released, it totally changed the game in terms of ETL processing on cloud storage.

Around this time, many of the customers we worked with were migrating their data analytics platforms (on-premises data warehouses) to Databricks in an effort to modernize their data stack. Databricks SQL was not available so many customers would ingest data and transform it using Databricks and

Parquet/Delta, but when it was time to report or integrate data with other systems the data was often replicated to a data warehouse or application database. These data warehouse migration projects started coming with velocity and as a consultant I was involved with several migration projects between the middle of 2018 and end of 2020.

As these projects grew in number there were obvious development patterns and best practices being discovered. One that seemed to work very well, and still does, is the idea of a metadata driven ETL processes. Where each table is registered in a look up table that a job orchestration tool can leverage to dynamically look up job parameters and execute code.

Let's assume you have a domain of 100 datasets. Each of these datasets are being ingested as json files and landing into a raw data zone within a cloud storage account on a regular cadence. The 100 datasets can be categorized as: append-only, truncate and replace, and merge load patterns. Parameterizing your code will allow you to avoid writing 100 different notebooks and instead allows you to write 3 notebooks that can be scheduled with different parameters. Parameterizing notebooks using Databricks Widgets or environment variables made the process of loading and working with a large number of tables simple, however, scheduling the jobs required outside tooling and adding dependent tasks after the initial data load was difficult to add on a per table basis which made this solution less than ideal. Improvements in the product since 2018 have drastically changed the way Databricks users develop and deploy data applications e.g. Databricks workflows allows for a native orchestration service instead of using a separate tool.

Now with Delta Live Tables (DLT) engineers can easily develop and deploy production grade data pipelines as scale. The value of DLT is extremely high

for SQL users who want to easily orchestrate and load data into target schemas. The Python syntax in DLT has always seemed a little more complicated to me when compared to the simplicity of SQL but does provide more enhanced operations for the engineers. The one area of the Python APIs for DLT that is (to say the least) **awesome**, is the ability to programmatically load many tables with minimal code. This is a perfect solution to the scenario we have discussed so far.

In this blog we will cover how to dynamically generate tables in Databricks using Python and Delta Live Tables. Associated code is available here.

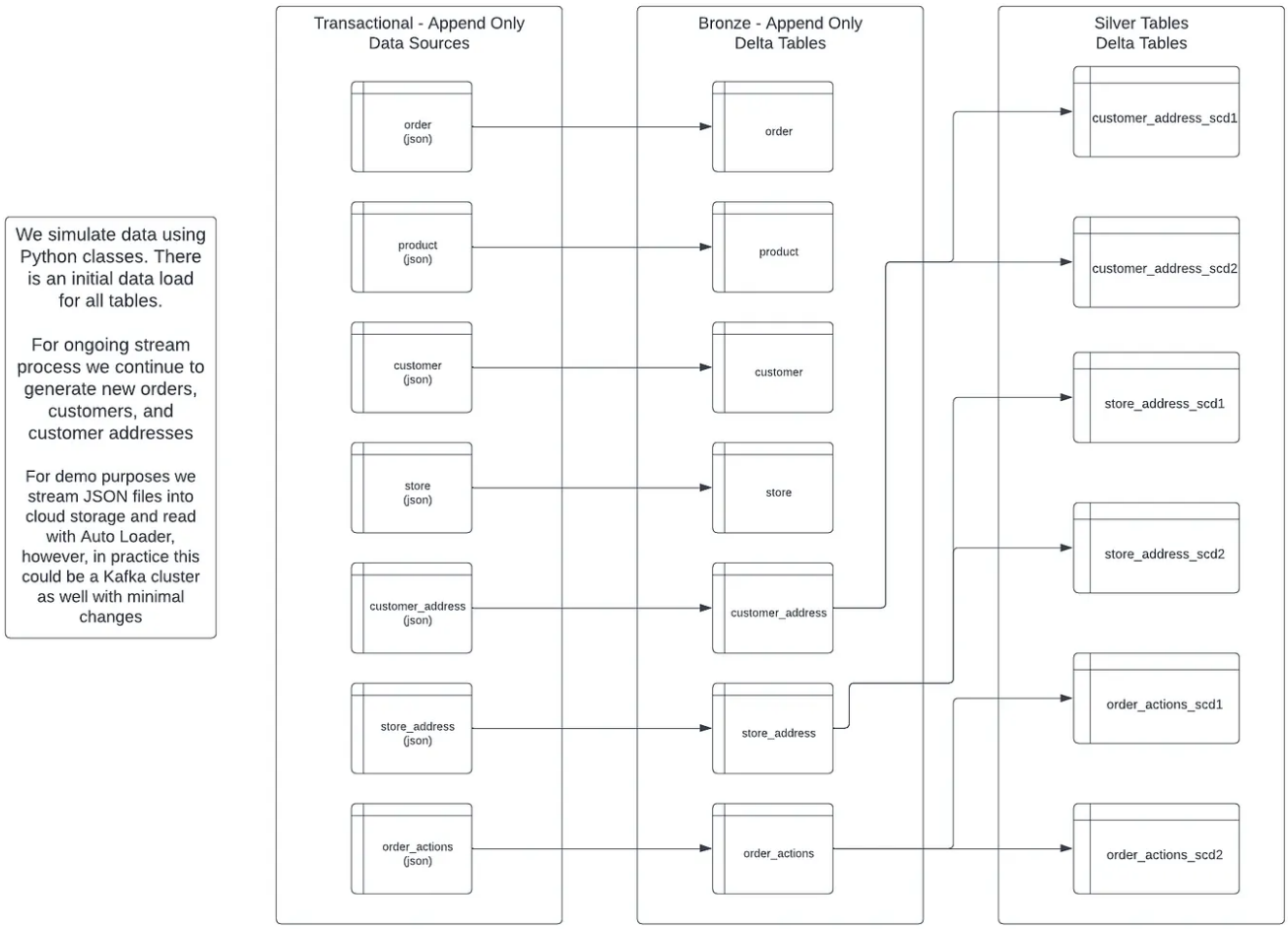
Working Example

To simplify our scenario, we will reduce the number of tables. We will leverage Delta Live Tables combined with Databricks Auto Loader to ingest seven different json data sources. The data we will be working with is simulated online orders for a retail company.

To generate data please run the GenerateData notebook. This will create the following tables:

- customer
- store
- customer_address
- store_address
- order
- product
- order_actions

The data generation notebook will, by default, run for approximately one hour before shutting down. It will begin by pre-populating a set of historical data as json files that we will want to load into delta tables using DLT. Once the initial dataset is created the notebook will continue to simulate customer behavior by randomly creating online orders and periodically updating customer addresses. Our goal is to create something like the following:



DLT Pipeline Data Flow

Delta Live Tables can be both continuous and triggered. When running continuously DLT will automatically load new data files as they are created. If running as triggered, then DLT will only load new files when the pipeline is executed. In both cases we will only load files that have not been previously processed due to the nature of Auto Loader.

To load the seven json datasets into Delta Lake using DLT you may think that you need to define each table individually, as shown below:

```
@dlt.table
def customer():
  return (
    spark.readStream.format('cloudfiles')
        .option('cloudFiles.format', 'json')
.load(f'/Users/customer/dynamic_dlt/raw/customer/customer_*.json')
.withColumn('input_file', input_file_name())
.withColumn("load_datetime", current_timestamp())

)

@dlt.table
def store():
  return (
    spark.readStream.format('cloudfiles')
        .option('cloudFiles.format', 'json')
.load(f'/Users/store/dynamic_dlt/raw/store/store_*.json')
.withColumn('input_file', input_file_name())
.withColumn("load_datetime", current_timestamp())

)

.
.
.
... Each table is defined separately
.
.

@dlt.table
def order_actions():
  return (
    spark.readStream.format('cloudfiles')
        .option('cloudFiles.format', 'json')
.load(f'/Users/order_actions/dynamic_dlt/raw/order_actions/order_acti
ons_*.json')
.withColumn('input_file', input_file_name())
.withColumn("load_datetime", current_timestamp())

)
```

However, in hopes of not repeating yourself (DRY principal) developers can do something like the following to reduce lines of code and simplify the data

pipeline. You will notice that we have two functions: `generate_tables` and `generate_scd_tables` . These are responsible for parameterizing two different types of tables. The first generates append only Delta tables in a bronze data layer, while the former creates slowly changing dimension tables (type 1 and 2).

```
tables = ['customer', 'store', 'customer_address', 'store_address',
'order', 'product', 'order_actions']

###
# This creates append only tables for our bronze sources
# we can do further modeling in silver/gold layers
###

def generate_tables(table):
  @dlt.table(
    name=table,
    comment="BRONZE: {}".format(table)
  )
  def create_table():
    return (
      spark.readStream.format('cloudfiles')
        .option('cloudFiles.format', 'json')
.load(f'/Users/{table}/dynamic_dlt/raw/{user_name}/{table}_*.json')
.withColumn('input_file', input_file_name())
.withColumn("load_datetime", current_timestamp())
)

# for each table we pass it into the above function
for t in tables:
  generate_tables(t)
```

Next, we want to create type one and type two slowly changing dimension tables. These can also be generated dynamically using a function and passing the values in.

```
def generate_scd_tables(table, key, seq_col="created_date",
scd_type=1):
  dlt.create_streaming_live_table("{}_scd{}".format(table, scd_type))
```

```
    dlt.apply_changes(
          target = "{}_scd{}".format(table, scd_type),
          source = table,
          keys = [key],
          sequence_by = col(seq_col),
          stored_as_scd_type = scd_type
        )

    generate_scd_tables('store_address', 'address_id', 'created_date', 1)
    generate_scd_tables('store_address', 'address_id', 'created_date', 2)
    generate_scd_tables('customer_address', 'address_id', 'created_date',
    1)
    generate_scd_tables('customer_address', 'address_id', 'created_date',
    2)
    generate_scd_tables('order_actions', 'order_id', 'datetime', 1)
    generate_scd_tables('order_actions', 'order_id', 'datetime', 2)
```

Now we have created a DLT pipeline with minimal amount of coding!

Delta Live Table Pipeline

Conclusion

Using parameterized functions to dynamically create and load tables in Delta Live Tables is a great way to simplify data pipelines. To learn more, check out the Delta Live Tables cookbook documentation! Happy pipelining!

Disclaimer: these are my own thoughts and opinions and not a reflection of my employer

Databricks Dlt Delta Live Tables Delta Python

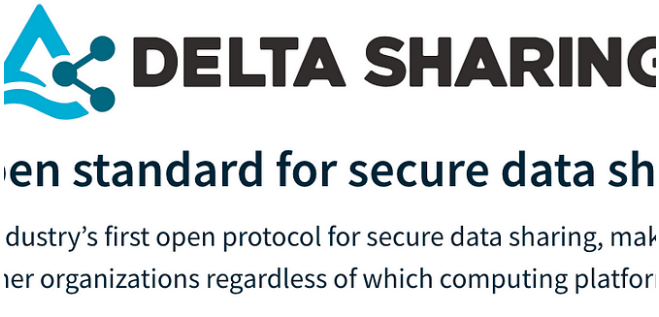
Written by Ryan Chynoweth

Edit profile

312 Followers

Senior Solutions Architect Databricks — anything shared is my own thoughts and opinions

More from Ryan Chynoweth



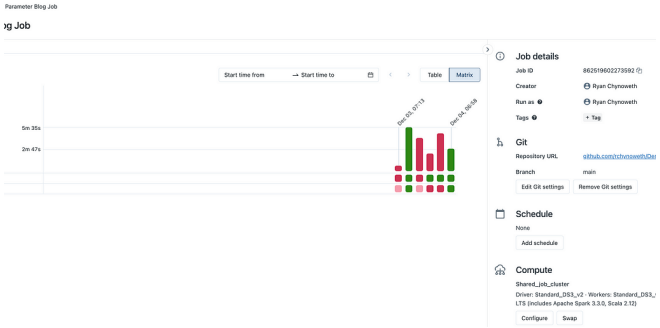
Ryan Chynoweth

Delta Sharing: An Implementation Guide for Multi-Cloud Architecture

Introduction

8 min read · 3 days ago

3 2



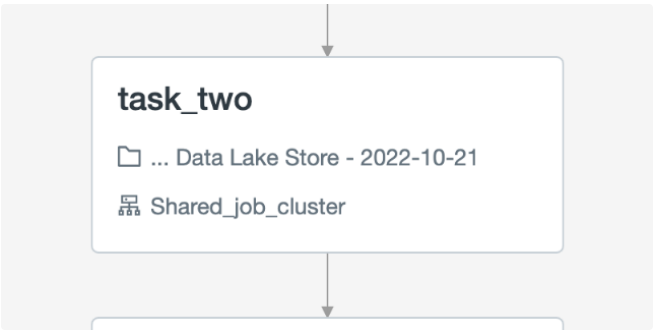
Ryan Chynoweth

Task Parameters and Values in Databricks Workflows

Databricks provides a set of powerful and dynamic orchestration capabilities that are...

11 min read · Dec 7, 2022

50 3



Ryan Chynoweth

Converting Stored Procedures to Databricks

Special thanks to co-author Kyle Hale, Sr. Specialist Solutions Architect at Databricks.

14 min read · Dec 29, 2022

116 5

Ryan Chynoweth

Recursive CTE on Databricks

Introduction

3 min read · Apr 20, 2022

33

See all from Ryan Chynoweth

Recommended from Medium

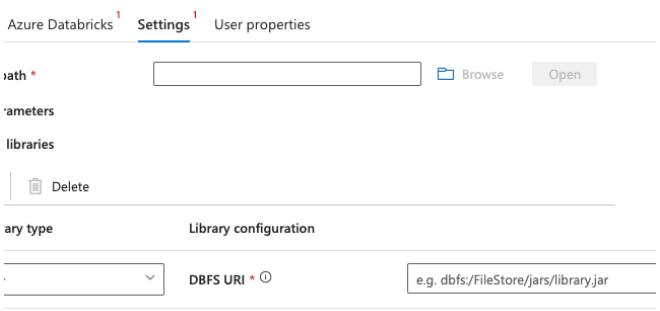
Daan Rademaker

Do-it-yourself, building your own Databricks Docker Container

In my previous LinkedIn article, I aimed to persuade you of the numerous advantages o...

7 min read · Oct 16, 2023

26



Matt Bradley

Azure Data Factory tips for running Databricks jobs

Following on from an earlier blog around using spot instances with Azure Data...

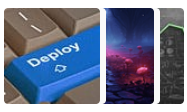
4 min read · Nov 2, 2023

14

Lists

Coding & Development

11 stories · 355 saves



Predictive Modeling w/ Python

20 stories · 749 saves

Practical Guides to Machine Learning

10 stories · 865 saves

ChatGPT

23 stories · 371 saves

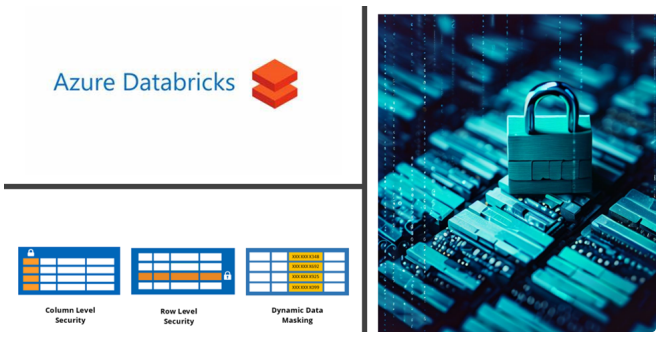
Nnaemezue Obi-Eyisi

Unveiling the Secrets: External Tables vs. External Volumes in...

While reviewing the Databricks documentation about Unity Catalog, I came...

· 7 min read · Sep 25, 2023

39 1



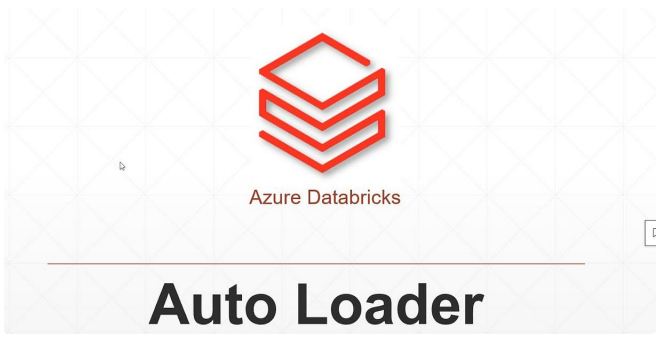
Samarendra Panda

Dynamic Row Level Filtering and Column Level Masking in Azure...

Background

5 min read · Sep 23, 2023

29

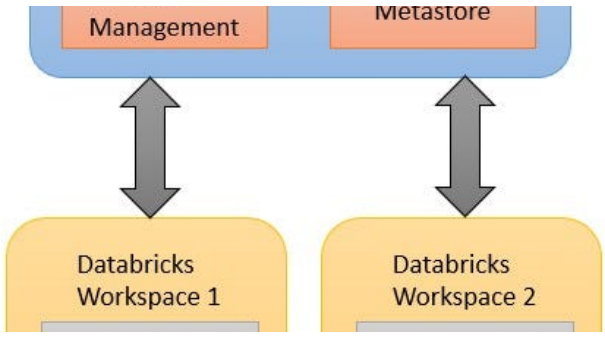


SIRIGIRI HARI KRISHNA in Towards Dev

Auto Loader

Autoloader simplifies reading various data file types from popular cloud locations like...

5 min read · Dec 9, 2023



Oindrila Chakraborty

Introduction to “Unity Catalog” in Databricks

What is “Unity Catalog”?

10 min read · Aug 14, 2023

See more recommendations