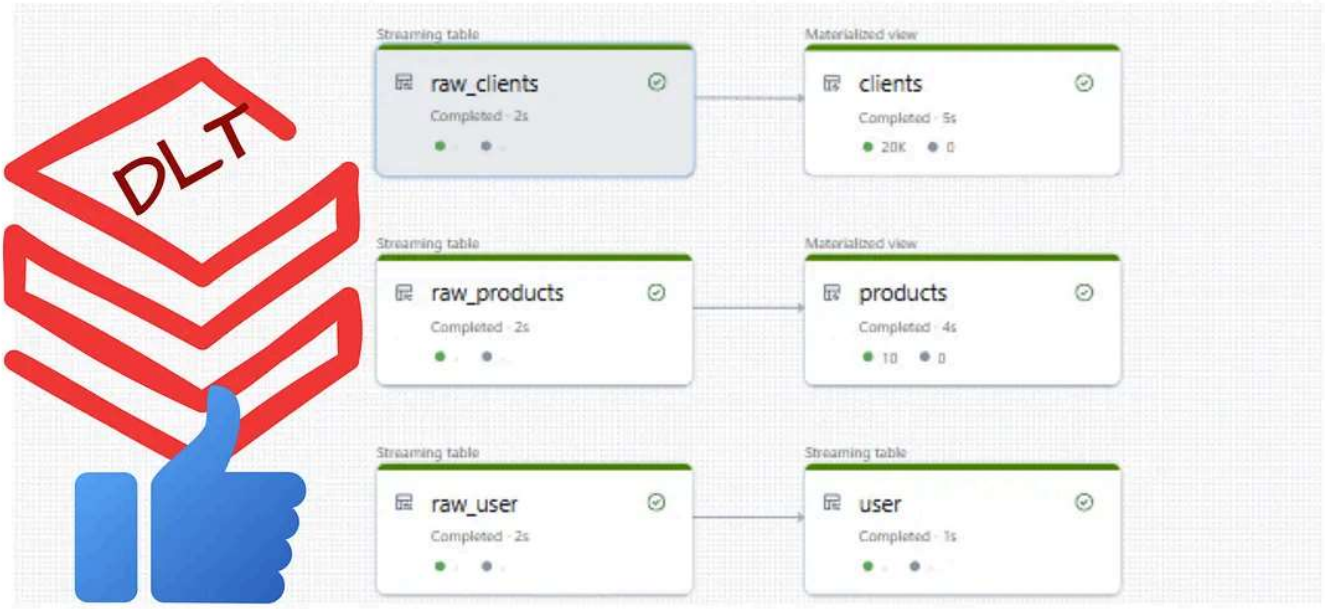
💡 Why I Liked Delta Live Tables in Databricks

Mariusz Kujawski Follow 7 min read · Feb 6, 2025

Delta Live Tables (DLT) is a **declarative framework** designed to simplify data ingestion processes in Databricks. Built on Apache Spark, DLT automates **orchestration, compute management, monitoring, and data quality**. While it has some limitations, recent enhancements have expanded its capabilities.

DLT allows you to create **streaming tables and materialized views**, which can be used for both **streaming and batch processing**.

In this post, I'll demonstrate how to build an **ingestion framework** that reads files from the **bronze layer** and populates tables in the **silver layer** — all in just a few steps. Let's dive in!



When to Use DLT

DLT is a **powerful tool** that accelerates ingestion processes, making it ideal when you need to quickly build a pipeline to deliver data to your **lakehouse.** It is particularly useful for:

- ✅ **Incremental data transformation**
  ✅ **Change Data Capture (CDC) support**
  ✅ **Type 2 Slowly Changing Dimensions (SCD2) handling out of the box**
- However, **DLT is not recommended for:**
- ❌ **Writing to external tables**
  ❌ **Complex transformations** that require extensive custom logic

Simple Example

To start working with **Delta Live Tables**, follow these steps:

1. **Create a Databricks notebook** and add the following sample code:

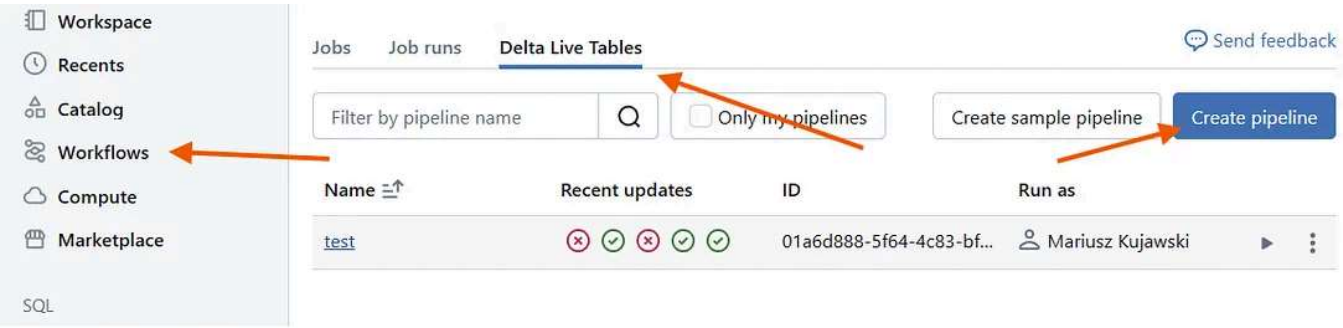
```
import dlt

@dlt.table(name=f"raw_products")
def read_raw():
    return (spark.readStream
    .format("cloudFiles")
    .option("cloudFiles.format", "csv")
    .option("inferSchema", True)
    .option("header", True)
    .load(f"abfss://bronze@xxxx.dfs.core.windows.net/products/")
    )


@dlt.table(name="products")
def read_silver():
    return spark.read.table(f"raw_products")
```

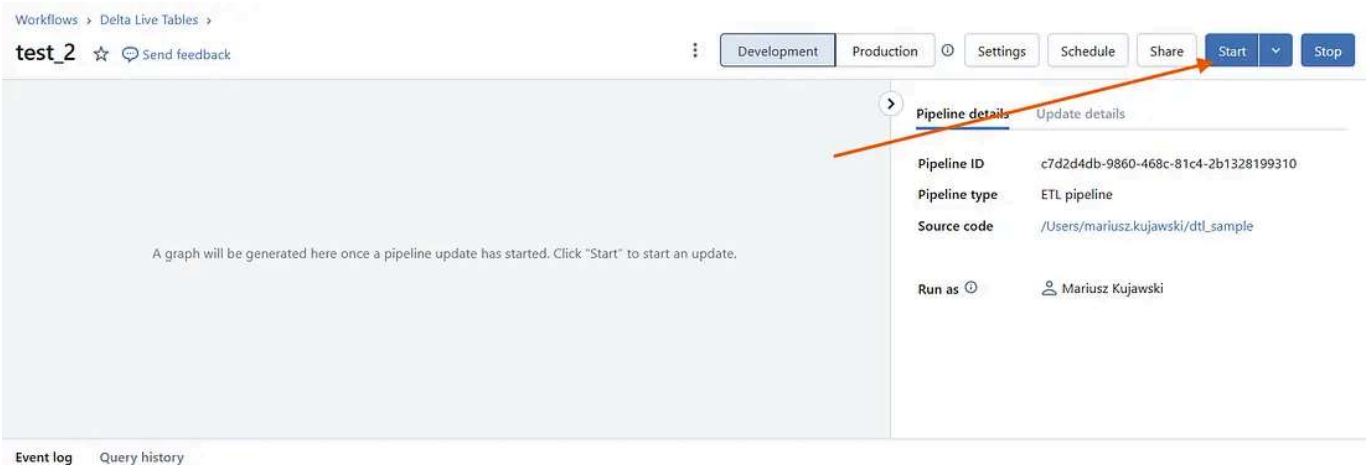
2. Set up a DLT pipeline:

- Navigate to **Workflows** > **Delta Live Tables** > **Create Pipeline**.
- In the form, enter a **pipeline name** and select your **notebook**.
- Choose a **default catalog and schema** for table creation.
- Click the **Create** button.



3. Run the pipeline:

- Click the **Start** button to create tables and execute the pipeline.



- In a few minutes, you should see the new tables in **Unity Catalog** along with the execution results.

Now, your pipeline is ready to populate tables! You can query the newly created tables from a **SQL Notebook** or a **Databricks notebook.**

Processing Multiple Tables with Delta Live Tables

As previously demonstrated, **DLT automates table creation and data loading**. However, one challenge in traditional ingestion frameworks is tracking which files have been processed. Typically, you'd need to implement a custom solution, **autoloader mechanism**, or **file archiving** to process only new files.

With **DLT, this logic is handled automatically**.

Scaling to Multiple Data Sources

A typical **lakehouse architecture** often involves ingesting data from **tens, hundreds, or even thousands** of sources. **DLT makes handling multiple sources more convenient**. Let's see how we can modify our code to process multiple sources dynamically.

```
import dlt

def gen_dlt(name):

    @dlt.table(name=f"raw_{name}")
    def read_raw():
        return (spark.readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "csv")
        .option("inferSchema", True)
        .option("header", True)
        .load(f"abfss://bronze@xxxx.dfs.core.windows.net/{name}/")
```

```
            )


        @dlt.table(name=f"silver.dbo.{name}")
        def read_silver():
            return spark.read.table(f"raw_{name}")

    tables = ["products", "clients"]

    for x in tables:
        gen_dlt(x)
```

The implementation is straightforward:

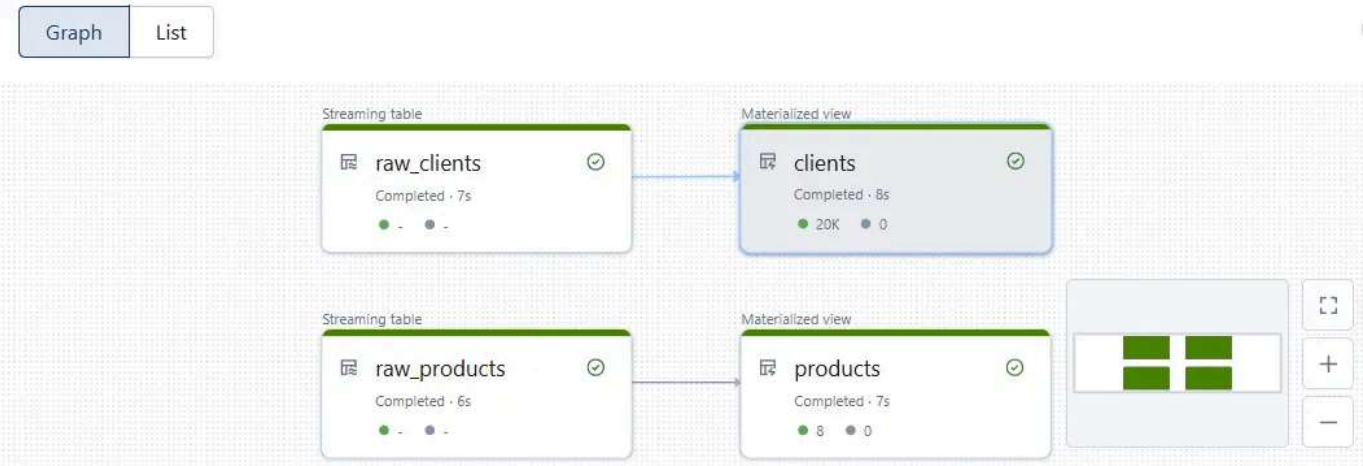
1. **Refactor your existing logic into a function body** (gen_dlt in my case)
2. **Parameterize** key elements such as:

- **DLT decorators**
- **Table names**
- **Data source paths**

3. **Use a configuration list** to manage multiple sources efficiently

- In this example, we define a `tables` variable containing a list of sources
- This list can also be **fetched from a YAML file or a metadata table**

As before, add the updated code to a **Databricks notebook** and configure a new **DLT pipeline**. The final result should look like this:



By leveraging a **configuration-driven approach**, you can create a **fully parameterized DLT pipeline** that seamlessly ingests data from the **bronze** layer into the **silver** layer. As you can see, **DLT simplifies multi-source ingestion** while maintaining flexibility.

Support for Multiple Catalogs and Schemas

Databricks recently introduced **multi-schema and multi-catalog support** for DLT. This allows you to better reflect **lakehouse architecture** by defining:

- **A separate schema for the bronze layer**
- **Another schema for the silver layer**

By default, tables are stored in the **catalog and schema defined in the DLT pipeline.** However, you can override this behavior using **explicit declarations.** The catalog and schema can also be **parameterized using a config file or metadata table.**

The following code snippet demonstrates how to specify a **catalog, schema, and table name** for a materialized view:

```
@dlt.table(name=f"silver.dbo.{name}")
    def read_silver():
        return spark.read.table(f"raw_{name}")
```

Applying Change Data Capture (CDC) with Delta Live Tables

Delta Live Tables (**DLT**) simplifies **Change Data Capture (CDC)** implementation and the **historization process** for the silver layer.

Depending on your **source data extraction strategy**, a common requirement is to store **historical changes** in a **silver table**. With **DLT**, we can easily implement **CDC, Slowly Changing Dimensions (SCD2), or historization** without complex transformations.

The APPLY CHANGES API

The `APPLY CHANGES` API provides a declarative approach to CDC and supports:

- ✅ **Tracking changes at the column level**
- ✅ **Defining a primary key**
- ✅ **Specifying CDC operations** (`INSERT`, `UPDATE`, `DELETE`)
- ✅ **Managing SCD types**

Modifying the Code for CDC Processing

Let's assume we have a **new source system** delivering data via a **CDC process.** To accommodate this, we need to adjust our configuration and modify the ingestion logic accordingly.

1. **Update the configuration list** to specify that the source uses CDC
2. **Modify the code** to dynamically adjust based on this parameter

```
import dlt
from pyspark.sql.functions import col, expr

def gen_dlt(name, cdc):

    @dlt.table(name=f"raw_{name}")
    def read_raw():
        return (spark.readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "csv")
        .option("inferSchema", True)
        .option("header", True)
        .load(f"abfss://bronze@xxx.dfs.core.windows.net/{name}/")
        )

    if not cdc:
        @dlt.table(name=f"silver.dbo.{name}")
        def read_silver():
            return spark.read.table(f"raw_{name}")
    else:
        dlt.create_streaming_table(f"silver.dbo.{name}")

        dlt.apply_changes(
        target = f"silver.dbo.{name}",
        source = f"raw_{name}",
        keys = ["userId"],
        sequence_by = col("sequenceNum"),
        apply_as_deletes = expr("operation = 'DELETE'"),
        except_column_list = ["operation", "sequenceNum"],
        stored_as_scd_type = 1
        )

tables = [
          {"source":"products", "cdc": False},
          {"source":"clients", "cdc": False},
```

```
            {"source":"user", "cdc": True}
            ]

for x in tables:
    gen_dlt(x["source"], x["cdc"])
```

With a **config-driven approach**, you can easily **adapt the pipeline** to handle various **load strategies** and **historization methods** by modifying the configuration table or file.

Processing SCD Type 2 with DLT

If you need to implement **SCD Type 2** logic, **DLT makes it incredibly simple**. You only need to **adjust a single parameter** in your code:

```
dlt.create_streaming_table(f"silver.dbo.{name}")

dlt.apply_changes(
  target = f"silver.dbo.{name}",
  source = f"raw_{name}",
  keys = ["userId"],
  sequence_by = col("sequenceNum"),
  apply_as_deletes = expr("operation = 'DELETE'"),
  except_column_list = ["operation", "sequenceNum"],
  stored_as_scd_type = 2
)
```

With this adjustment, DLT **automatically tracks historical changes**, maintaining previous records and adding new ones as updates occur.

Deletion Behavior of Materialized Views and Streaming Tables in DLT

One **potential drawback** of Delta Live Tables (**DLT**) is the **automatic deletion of tables** when you remove a **DLT pipeline**. If a table is removed from your **code or configuration file**, DLT will also delete it from **Unity Catalog**.

Fortunately, **Databricks recently introduced a feature** that allows you to **restore dropped tables and views**.

How to Restore Deleted Tables

If you accidentally delete a table, follow these steps to recover it:

1️⃣ List Dropped Tables:

```
SHOW TABLES DROPPED in dw
```

2️⃣ Restore a Table Using `UNDROP` Command:

```
UNDROP TABLE { table_name | WITH ID table_id }
```

With this feature, accidental deletions are no longer a major concern, as you can **quickly recover your tables** in just a few steps.

Delta Live Tables with SQL

Alternatively, you can define your DLT pipelines by creating streaming tables and materialized views using SQL. This method may be more convinced for

users skilled with SQL.

```
CREATE OR REFRESH STREAMING TABLE baby_names_raw
COMMENT "Popular baby first names in New York. This data was ingested from the N
AS SELECT Year, `First Name` AS First_Name, County, Sex, Count
FROM STREAM(read_files(
  '/Volumes/${my_catalog}/${my_schema}/${my_volume}/',
  format => 'csv',
  header => true,
  mode => 'FAILFAST'));


CREATE OR REFRESH MATERIALIZED VIEW baby_names_prepared(
  CONSTRAINT valid_first_name EXPECT (First_Name IS NOT NULL),
  CONSTRAINT valid_count EXPECT (Count > 0) ON VIOLATION FAIL UPDATE
)
COMMENT "New York popular baby first name data cleaned and prepared for analysis
AS SELECT
  Year AS Year_Of_Birth,
  First_Name,
  Count
FROM LIVE.baby_names_raw;
```

Summary

Delta Live Tables (**DLT**) may not be the best choice for every scenario, but as we've seen, it excels at **simplifying data ingestion** for use cases requiring **basic preprocessing** — such as **removing columns, adding metadata, and filtering data.**

With DLT, you can **build an ingestion pipeline in just a few hours**, compared to the **days** required to develop a custom framework. Additionally, **built-in data quality enforcement** provides an extra layer of reliability.

The **continuous improvements** and **new features** being added to DLT make it an increasingly powerful **tool** for data engineers.

If you found this article insightful, please click the 'clap' button and follow me on Medium and LinkedIn. For any questions or advice, feel free to reach out to me on LinkedIn.

Databricks | Data Science | Data Engineering | DIt

Written by Mariusz Kujawski

3.2K followers · 2 following

Cloud Data Architect | Data Engineer https://www.linkedin.com/in/mariusz-kujawski-812bb1103/

Follow

Responses (6)

Write a response

What are your thoughts?

Eric Wallace
Feb 13

This is one of the better articles written about DLT, thank you for this. Most of this information is sprinkled throughout various databricks docs and this kind of puts it all together

14 Reply

P

Patrick Farry
Jul 7

Hi Mariusz, this is a great article. I quoted you on an InfoQ news item about Databricks open sourcing DLT. https://www.infoq.com/news/2025/07/databricks-declarative-pipelines/

1 reply Reply

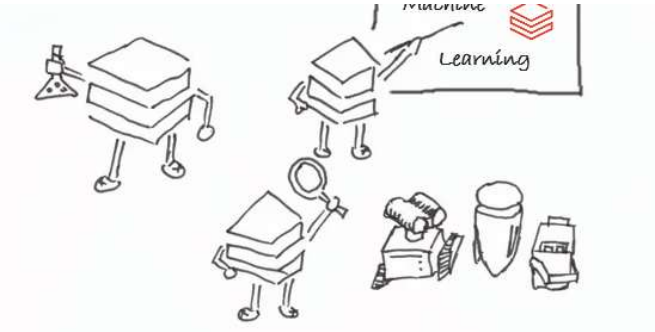
Alex Ott
Mar 20

It's also recommended to look to append flows - in some cases you may need to add another location with similar data that should be written into the same table, or do a backfill if you want to change existing data.

Reply

See all responses

More from Mariusz Kujawski

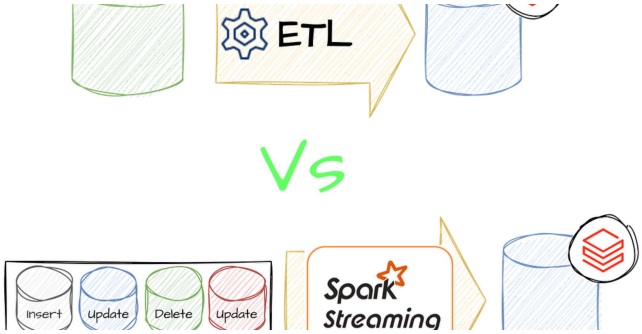


Mariusz Kujawski

Introduction to Databricks: A Beginner's Guide

In this guide, I'll walk you through everything you need to know to get started with...

Feb 26, 2024 542 12



Mariusz Kujawski

💡 Incremental Ingestion with CDC and Auto Loader: Streaming Isn't...

Did you know streaming isn't just for real-time? You can also use it to replace...

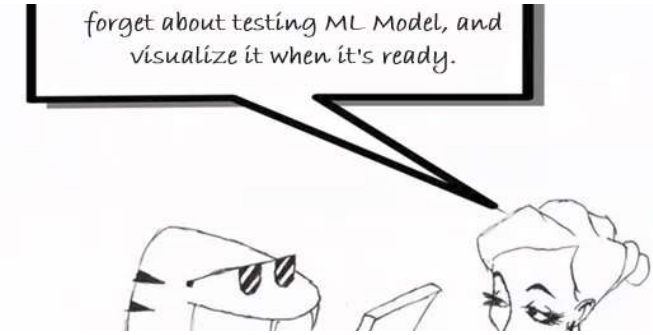
Jul 10 44 2

Mariusz Kujawski

🚀 Custom Data Lineage in Databricks

When I work with customers such as banks or insurance companies, one of the most...

Jun 30 94 1



Mariusz Kujawski

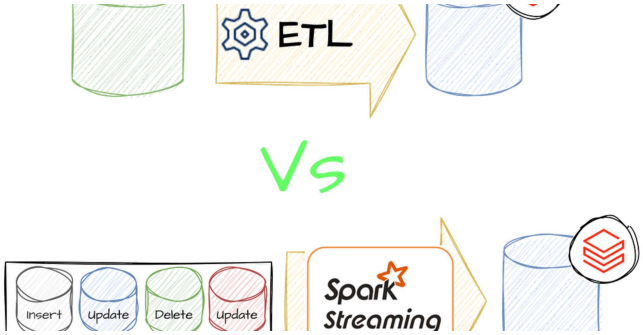
Python for Data Engineering

Python plays a crucial role in the world of data engineering, offering versatile and powerful...

Dec 4, 2023 799 9

See all from Mariusz Kujawski

Recommended from Medium



Mariusz Kujawski

💡 Incremental Ingestion with CDC and Auto Loader: Streaming Isn't...

Did you know streaming isn't just for real-time? You can also use it to replace...

Jul 10 44 2

In Tech with Abhishek by Abhishek Kumar Gupta

✅ 50 Databricks Tips, Tricks & Best Practices for 2025

Unlock productivity, performance, and cost savings with real-world Databricks wisdom.

Jun 30 96 3

Skye

EY : Senior Data Engineer Interview

All Questions That I Was Grilled On

Nnaemezue Obi-Eyisi

Databricks One Might Be the Most Important Shift Yet—Here's Why...

I recently attended the Databricks Data + AI Summit (virtually), and among all the buzz...

Jul 5 151 3

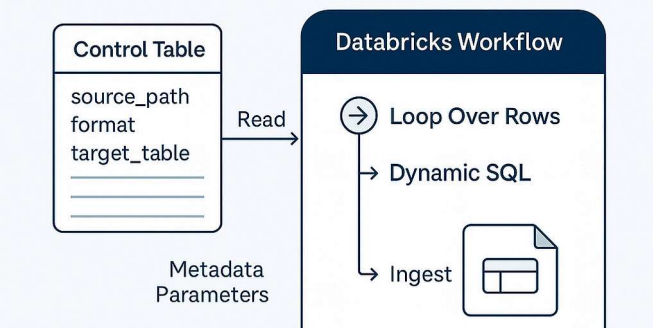
In Towards Data Engineering by Avinash Jha

How to Calculate Jobs, Stages, and Tasks in Apache Spark

If you're learning Apache Spark or preparing for a data engineer interview, understanding...

Jul 24 84 1

Jun 21 28 3



In DBSQL SME Engineering by Franco Patano

A primer for metadata-driven frameworks with Databricks...

Metadata-driven frameworks are a proven way to control data ingestion at scale. Instea...

Apr 29 106 5

See more recommendations