Delta Live Tables

Delta Live Tables is a declarative framework for building reliable, maintainable, and testable data processing pipelines. You define the transformations to perform on your data and Delta Live Tables manages task orchestration, cluster management, monitoring, data quality, and error handling.

Note

Delta Live Tables requires the **Premium plan**. Contact your Databricks account team for more information.

Instead of defining your data pipelines using a series of separate Apache Spark tasks, you define streaming tables and materialized views that the system should create and keep up to date. Delta Live Tables manages how your data is transformed based on queries you define for each processing step. You can also enforce data quality with Delta Live Tables *expectations*, which allow you to define expected data quality and specify how to handle records that fail those expectations.

What are Delta Live Tables datasets?

Delta Live Tables datasets are the streaming tables, materialized views, and views maintained as the results of declarative queries. The following table describes how each dataset is processed:

Expand table

Dataset type	**How are records processed through defined queries?**
Streaming table	Each record is processed exactly once. This assumes an append-only source.
Materialized views	Records are processed as required to return accurate results for the current data state. Materialized views should be used for data sources with updates, deletions, or aggregations, and for change data capture processing (CDC).
Views	Records are processed each time the view is queried. Use views for intermediate transformations and data quality checks that should not be published to public datasets.

The following sections provide more detailed descriptions of each dataset type.

Streaming table

A *streaming table* is a Delta table with extra support for streaming or incremental data processing. Streaming tables allow you to process a growing dataset, handling each row only once. Because most datasets grow continuously over time, streaming tables are good for most ingestion workloads. Streaming tables are optimal for pipelines that require data freshness and low latency. Streaming tables can also be useful for massive scale transformations, as results can be incrementally calculated as new data arrives, keeping results up to date without needing to fully recompute all source data with each update. Streaming tables are designed for data sources that are append-only.

Note

Although, by default, streaming tables require append-only data sources, when a streaming source is another streaming table that requires updates or deletes, you can override this behavior with the **skipChangeCommits flag**.

Materialized view

A *materialized view* (or *live table*) is a view where the results have been precomputed. Materialized views are refreshed according to the update schedule of the pipeline in which they're contained. Materialized views are powerful because they can handle any changes in the input. Each time the pipeline updates, query results are recalculated to reflect changes in upstream datasets that might have occurred because of compliance, corrections, aggregations, or general CDC. Delta Live Tables implements materialized views as Delta tables, but abstracts away complexities associated with efficient application of updates, allowing users to focus on writing queries.

Views

All *views* in Azure Databricks compute results from source datasets as they are queried, leveraging caching optimizations when available. Delta Live Tables does not publish views to the catalog, so views can be referenced only within the pipeline in which they are defined. Views are useful as intermediate queries that should not be exposed to end users or systems. Databricks recommends using views to enforce data quality constraints or transform and enrich datasets that drive multiple downstream queries.

Declare your first datasets in Delta Live Tables

Delta Live Tables introduces new syntax for Python and SQL.

Note

Delta Live Tables separates dataset definitions from update processing, and Delta Live Tables notebooks are not intended for interactive execution.

What is a Delta Live Tables pipeline?

A *pipeline* is the main unit used to configure and run data processing workflows with Delta Live Tables.

A pipeline contains materialized views and streaming tables declared in Python or SQL source files. Delta Live Tables infers the dependencies between these tables, ensuring updates occur in the correct order. For each dataset, Delta Live Tables compares the current state with the desired state and proceeds to create or update datasets using efficient processing methods.

The settings of Delta Live Tables pipelines fall into two broad categories:

1. Configurations that define a collection of notebooks or files (known as *source code* or *libraries*) that use Delta Live Tables syntax to declare datasets.
2. Configurations that control pipeline infrastructure, dependency management, how updates are processed, and how tables are saved in the workspace.

Most configurations are optional, but some require careful attention, especially when configuring production pipelines. These include the following:

- To make data available outside the pipeline, you must declare a **target schema** to publish to the Hive metastore or a **target catalog** and **target schema** to publish to Unity Catalog.
- Data access permissions are configured through the cluster used for execution. Make sure your cluster has appropriate permissions configured for data sources and the target **storage location**, if specified.

Deploy your first pipeline and trigger updates

Before processing data with Delta Live Tables, you must configure a pipeline. Once a pipeline is configured, you can trigger an update to calculate results for each dataset in your pipeline.

What is a pipeline update?

Pipelines deploy infrastructure and recompute data state when you start an *update*. An update does the following:

- Starts a cluster with the correct configuration.
- Discovers all the tables and views defined, and checks for any analysis errors such as invalid column names, missing dependencies, and syntax errors.
- Creates or updates tables and views with the most recent data available.

Pipelines can be run continuously or on a schedule depending on your use case's cost and latency requirements.

Ingest data with Delta Live Tables

Delta Live Tables supports all data sources available in Azure Databricks.

Databricks recommends using streaming tables for most ingestion use cases. For files arriving in cloud object storage, Databricks recommends Auto Loader. You can directly ingest data with Delta Live Tables from most message buses.

For formats not supported by Auto Loader, you can use Python or SQL to query any format supported by Apache Spark.

Monitor and enforce data quality

You can use *expectations* to specify data quality controls on the contents of a dataset. Unlike a CHECK constraint in a traditional database which prevents adding any records that fail the constraint, expectations provide flexibility when processing data that fails data quality requirements. This flexibility allows you to process and store data that you expect to be messy and data that must meet strict quality requirements.

How are Delta Live Tables and Delta Lake related?

Delta Live Tables extends the functionality of Delta Lake. Because tables created and managed by Delta Live Tables are Delta tables, they have the same guarantees and features provided by Delta Lake.

Delta Live Tables adds several table properties in addition to the many table properties that can be set in Delta Lake.

How tables are created and managed by Delta Live Tables

Azure Databricks automatically manages tables created with Delta Live Tables, determining how updates need to be processed to correctly compute the current state of a table and performing a number of maintenance and optimization tasks.

For most operations, you should allow Delta Live Tables to process all updates, inserts, and deletes to a target table.

Maintenance tasks performed by Delta Live Tables

Delta Live Tables performs maintenance tasks within 24 hours of a table being updated. Maintenance can improve query performance and reduce cost by removing old versions of tables. By default, the system performs a full OPTIMIZE operation followed by VACUUM. You can disable OPTIMIZE for a table by setting pipelines.autoOptimize.managed = false in the table properties for the table. Maintenance tasks are performed only if a pipeline update has run in the 24 hours before the maintenance tasks are scheduled.

Limitations

The following limitations apply:

- All tables created and updated by Delta Live Tables are Delta tables.

- Delta Live Tables tables can only be defined once, meaning they can only be the target of a single operation in all Delta Live Tables pipelines.
- Identity columns are not supported with tables that are the target of APPLY CHANGES INTO and might be recomputed during updates for materialized views. For this reason, Databricks recommends only using identity columns with streaming tables in Delta Live Tables.
- An Azure Databricks workspace is limited to 100 concurrent pipeline updates.

Tutorial: Run your first Delta Live Tables pipeline

To demonstrate Delta Live Tables functionality, the examples in this tutorial download a publicly available dataset. However, Databricks has several ways to connect to data sources and ingest data that pipelines implementing real-world use cases will use.

Requirements

- To start a non-serverless pipeline, you must have cluster creation permission or access to a cluster policy defining a Delta Live Tables cluster. The Delta Live Tables runtime creates a cluster before it runs your pipeline and fails if you don't have the correct permission.
- To use the examples in this tutorial, your workspace must have Unity Catalog enabled.
- You must have the following permissions in Unity Catalog:
  - READ VOLUME and WRITE VOLUME, or ALL PRIVILEGES, for the my-volume volume.
  - USE SCHEMA or ALL PRIVILEGES for the default schema.
  - USE CATALOG or ALL PRIVILEGES for the main catalog.
- The examples in this tutorial use a Unity Catalog volume to store sample data. To use these examples, create a volume and use that volume's catalog, schema, and volume names to set the volume path used by the examples.

Note

If your workspace does not have Unity Catalog enabled, **notebooks** parveenkrraina/Databricks-Unity-Catalog (github.com) with examples that do not require Unity Catalog are attached to this article. To use these examples, select Hive metastore as the storage option when you create the pipeline.

Where do you run Delta Live Tables queries?

Delta Live Tables queries are primarily implemented in Databricks notebooks, but Delta Live Tables is not designed to be run interactively in notebook cells. Executing a cell that contains Delta Live Tables syntax in a Databricks notebook results in an error message. To run your queries, you must configure your notebooks as part of a pipeline.

Important

- You cannot rely on the cell-by-cell execution ordering of notebooks when writing queries for Delta Live Tables. Delta Live Tables evaluates and runs all code defined in notebooks but has a different execution model than a notebook **Run all** command.

- You cannot mix languages in a single Delta Live Tables source code file. For example, a notebook can contain only Python queries or SQL queries. If you must use multiple languages in a pipeline, use multiple language-specific notebooks or files in the pipeline.

You can also use Python code stored in files. For example, you can create a Python module that can be imported into your Python pipelines or define Python user-defined functions (UDFs) to use in SQL queries.

Example: Ingest and process New York baby names data

The example in this article uses a publicly available dataset that contains records of New York State baby names. These examples demonstrate using a Delta Live Tables pipeline to:

- Read raw CSV data from a publicly available dataset into a table.
- Read the records from the raw data table and use Delta Live Tables expectations to create a new table that contains cleansed data.
- Use the cleansed records as input to Delta Live Tables queries that create derived datasets.

This code demonstrates a simplified example of the medallion architecture

Implementations of this example are provided for the Python and SQL interfaces. You can follow the steps to create new notebooks that contain the example code, or you can skip ahead to Create a pipeline and use one of the notebooks provided on this page.

Implement a Delta Live Tables pipeline with Python

Python code that creates Delta Live Tables datasets must return DataFrames, familiar to users with PySpark or Pandas for Spark experience. For users unfamiliar with DataFrames, Databricks recommends using the SQL interface.

All Delta Live Tables Python APIs are implemented in the dlt module. Your Delta Live Tables pipeline code implemented with Python must explicitly import the dlt module at the top of Python notebooks and files. Delta Live Tables differs from many Python scripts in a key way: you do not call the functions that perform data ingestion and transformation to create Delta Live Tables datasets. Instead, Delta Live Tables interprets the decorator functions from the dlt module in all files loaded into a pipeline and builds a dataflow graph.

To implement the example in this tutorial, copy and paste the following Python code into a new Python notebook. You should add each example code snippet to its own cell in the notebook in the order described.

Note

When you create a pipeline with the Python interface, by default, table names are defined by function names. For example, the following Python example creates three tables named baby_names_raw, baby_names_prepared, and top_baby_names_2021. You can override the table name using the name parameter.

Import the Delta Live Tables module

All Delta Live Tables Python APIs are implemented in the dlt module. Explicitly import the dlt module at the top of Python notebooks and files.

The following example shows this import, alongside import statements for pyspark.sql.functions.

Python

```
import dlt

from pyspark.sql.functions import *
```

Download the data

To get the data for this example, you download a CSV file and store it in the volume as follows:

Python

```
import os


os.environ["UNITY_CATALOG_VOLUME_PATH"] = "/Volumes/<catalog-name>/<schema-name>/<volume-name>/"

os.environ["DATASET_DOWNLOAD_URL"] = "https://health.data.ny.gov/api/views/jxy9-yhdk/rows.csv"

os.environ["DATASET_DOWNLOAD_FILENAME"] = "rows.csv"


dbutils.fs.cp(f"{os.environ.get('DATASET_DOWNLOAD_URL')}", f"{os.environ.get('UNITY_CATALOG_VOLUME_PATH')}{os.environ.get('DATASET_DOWNLOAD_FILENAME')}")
```

Replace <catalog-name>, <schema-name>, and <volume-name> with the catalog, schema, and volume names for a Unity Catalog volume.

Create a table from files in object storage

Delta Live Tables supports loading data from all formats supported by Azure Databricks. See Data format options.

The @dlt.table decorator tells Delta Live Tables to create a table that contains the result of a DataFrame returned by a function. Add the @dlt.table decorator before any Python function definition that returns a Spark DataFrame to register a new table in Delta Live Tables. The following example demonstrates using the function name as the table name and adding a descriptive comment to the table:

Python

```
@dlt.table(
  comment="Popular baby first names in New York. This data was ingested from the New York State Department of Health."
)
def baby_names_raw():
  df = spark.read.csv(f"{os.environ.get('UNITY_CATALOG_VOLUME_PATH')}{os.environ.get('DATASET_DOWNLOAD_FILENAME')}", header=True, inferSchema=True)
  df_renamed_column = df.withColumnRenamed("First Name", "First_Name")
  return df_renamed_column
```

Add a table from an upstream dataset in the pipeline

You can use dlt.read() to read data from other datasets declared in your current Delta Live Tables pipeline. Declaring new tables in this way creates a dependency that Delta Live Tables automatically resolves before executing updates.

Python

```
@dlt.table(
  comment="New York popular baby first name data cleaned and prepared for analysis."
)
@dlt.expect("valid_first_name", "First_Name IS NOT NULL")
@dlt.expect_or_fail("valid_count", "Count > 0")
def baby_names_prepared():
  return (
```

```python
    dlt.read("baby_names_raw")
      .withColumnRenamed("Year", "Year_Of_Birth")
      .select("Year_Of_Birth", "First_Name", "Count")
  )
```

Create a table with enriched data views

Because Delta Live Tables processes updates to pipelines as a series of dependency graphs, you can declare highly enriched views that power dashboards, BI, and analytics by declaring tables with specific business logic.

Tables in Delta Live Tables are equivalent conceptually to materialized views. Whereas traditional views on Spark run logic each time the view is queried, a Delta Live Tables table stores the most recent version of query results in data files. Because Delta Live Tables manages updates for all datasets in a pipeline, you can schedule pipeline updates to match latency requirements for materialized views and know that queries against these tables contain the most recent version of data available.

The table defined by the following code demonstrates the conceptual similarity to a materialized view derived from upstream data in your pipeline:

Python

```python
@dlt.table(
  comment="A table summarizing counts of the top baby names for New York for 2021."
)
def top_baby_names_2021():
  return (
    dlt.read("baby_names_prepared")
      .filter(expr("Year_Of_Birth == 2021"))
      .groupBy("First_Name")
      .agg(sum("Count").alias("Total_Count"))
      .sort(desc("Total_Count"))
      .limit(10)
  )
```

To configure a pipeline that uses the notebook, see Create a pipeline.

Implement a Delta Live Tables pipeline with SQL

Databricks recommends Delta Live Tables with SQL as the preferred way for SQL users to build new ETL, ingestion, and transformation pipelines on Azure Databricks. The SQL interface for Delta Live Tables extends standard Spark SQL with many new keywords, constructs, and table-valued functions. These additions to standard SQL allow users to declare dependencies between datasets and deploy production-grade infrastructure without learning new tooling or additional concepts.

For users familiar with Spark DataFrames and who need support for more extensive testing and operations that are difficult to implement with SQL, such as metaprogramming operations, Databricks recommends using the Python interface. See [Example: Ingest and process New York baby names data]().

Download the data

To get the data for this example, copy the following code, paste it into a new notebook, and then run the notebook. To review options for creating notebooks, see [Create a notebook]().

Bash

```
%sh

wget -O "/Volumes/<catalog-name>/<schema-name>/<volume-name>/babynames.csv" "https://health.data.ny.gov/api/views/jxy9-yhdk/rows.csv"
```

Replace <catalog-name>, <schema-name>, and <volume-name> with the catalog, schema, and volume names for a Unity Catalog volume.

Create a table from files in Unity Catalog

For the rest of this example, copy the following SQL snippets and paste them into a new SQL notebook, separate from the notebook in the previous section. You should add each example SQL snippet to its own cell in the notebook in the order described.

Delta Live Tables supports loading data from all formats supported by Azure Databricks. See [Data format options]().

All Delta Live Tables SQL statements use CREATE OR REFRESH syntax and semantics. When you update a pipeline, Delta Live Tables determines whether the logically correct result for the table can be accomplished through incremental processing or if full recomputation is required.

The following example creates a table by loading data from the CSV file stored in the Unity Catalog volume:

SQL

```
CREATE OR REFRESH LIVE TABLE baby_names_sql_raw

COMMENT "Popular baby first names in New York. This data was ingested from the New York State Department of Health."

AS SELECT Year, `First Name` AS First_Name, County, Sex, Count FROM read_files(

  '/Volumes/<catalog-name>/<schema-name>/<volume-name>/babynames.csv',

  format => 'csv',

  header => true,

  mode => 'FAILFAST')
```

Replace <catalog-name>, <schema-name>, and <volume-name> with the catalog, schema, and volume names for a Unity Catalog volume.

Add a table from an upstream dataset to the pipeline

You can use the live virtual schema to query data from other datasets declared in your current Delta Live Tables pipeline. Declaring new tables in this way creates a dependency that Delta Live Tables automatically resolves before executing updates. The live schema is a custom keyword implemented in Delta Live Tables that can be substituted for a target schema if you want to publish your datasets. See Use Unity Catalog with your Delta Live Tables pipelines and Publish data from Delta Live Tables pipelines to the Hive metastore.

The following code also includes examples of monitoring and enforcing data quality with expectations. See Manage data quality with Delta Live Tables.

SQL

```
CREATE OR REFRESH LIVE TABLE baby_names_sql_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_sql_raw;
```

Create an enriched data view

Because Delta Live Tables processes updates to pipelines as a series of dependency graphs, you can declare highly enriched views that power dashboards, BI, and analytics by declaring tables with specific business logic.

Live tables are equivalent conceptually to materialized views. Whereas traditional views on Spark run logic each time the view is queried, live tables store the most recent version of query results in data files. Because Delta Live Tables manages updates for all datasets in a pipeline, you can schedule pipeline updates to match latency requirements for materialized views and know that queries against these tables contain the most recent version of data available.

The following code creates an enriched materialized view of upstream data:

SQL

```
CREATE OR REFRESH LIVE TABLE top_baby_names_sql_2021

COMMENT "A table summarizing counts of the top baby names for New York for 2021."

AS SELECT

  First_Name,

  SUM(Count) AS Total_Count

FROM live.baby_names_sql_prepared

WHERE Year_Of_Birth = 2021

GROUP BY First_Name

ORDER BY Total_Count DESC

LIMIT 10;
```

To configure a pipeline that uses the notebook, continue to Create a pipeline.

Create a pipeline

Delta Live Tables creates pipelines by resolving dependencies defined in notebooks or files (called *source code* or *libraries*) using Delta Live Tables syntax. Each source code file can only contain one language, but you can mix libraries of different languages in your pipeline.

1. Click **Delta Live Tables** in the sidebar and click **Create Pipeline**.

2. Give the pipeline a name.

3. (Optional) Select the **Serverless** checkbox to use fully managed compute for this pipeline. When you select **Serverless**, the **Compute** settings are removed from the UI.
4. (Optional) Select a product edition.
5. Select **Triggered** for **Pipeline Mode**.
6. Configure one or more notebooks containing the source code for the pipeline. In the **Paths** textbox, enter the path to a notebook or click to select a notebook.
7. Select a destination for datasets published by the pipeline, either the Hive metastore or Unity Catalog.
    - **Hive metastore**:
        - (Optional) Enter a **Storage location** for output data from the pipeline. The system uses a default location if you leave **Storage location** empty.
        - (Optional) Specify a **Target schema** to publish your dataset to the Hive metastore.
    - **Unity Catalog**: Specify a **Catalog** and a **Target schema** to publish your dataset to Unity Catalog.
8. (Optional) If you have not selected **Serverless**, you can configure compute settings for the pipeline.
9. (Optional) Click **Add notification** to configure one or more email addresses to receive notifications for pipeline events.
10. (Optional) Configure advanced settings for the pipeline.
11. Click **Create**.

The system displays the **Pipeline Details** page after you click **Create**. You can also access your pipeline by clicking the pipeline name in the **Delta Live Tables** tab.

Start a pipeline update

To start an update for a pipeline, click the Start button in the top panel. The system returns a message confirming that your pipeline is starting.

After successfully starting the update, the Delta Live Tables system:

1. Starts a cluster using a cluster configuration created by the Delta Live Tables system. You can also specify a custom cluster configuration.
2. Creates any tables that don't exist and ensures that the schema is correct for any existing tables.
3. Updates tables with the latest data available.
4. Shuts down the cluster when the update is complete.

Note

Execution mode is set to **Production** by default, which deploys ephemeral compute resources for each update. You can use **Development** mode to change this behavior, allowing the same compute resources to be used for multiple pipeline updates during development and testing.

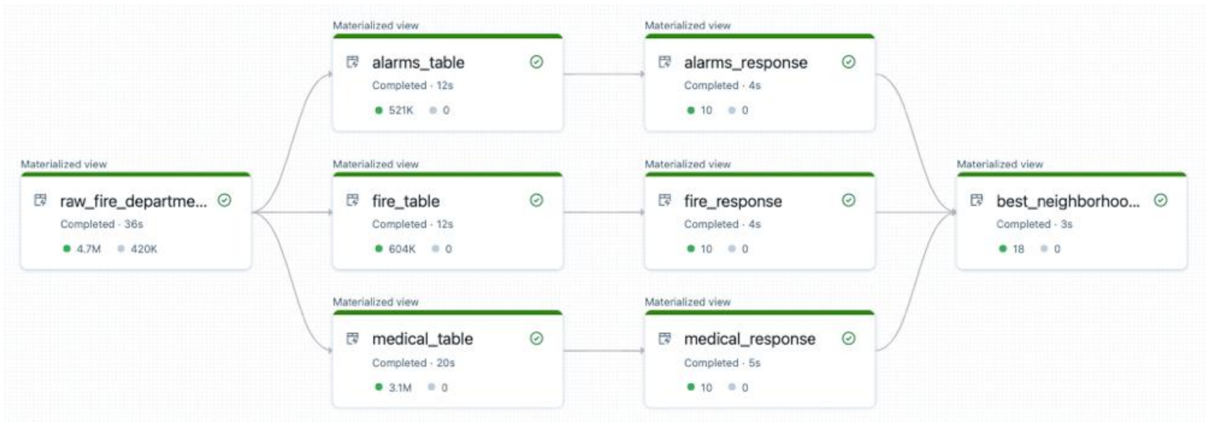
Publish datasets

You can make Delta Live Tables datasets available for querying by publishing tables to the Hive metastore or Unity Catalog. If you do not specify a target for publishing data, tables created in Delta Live Tables pipelines can only be accessed by other operations in that same pipeline.

Programmatically create multiple tables

You can use Python with Delta Live Tables to programmatically create multiple tables to reduce code redundancy.

You might have pipelines containing multiple flows or dataset definitions that differ only by a small number of parameters. This redundancy results in pipelines that are error-prone and difficult to maintain. For example, the following diagram shows the graph of a pipeline that uses a fire department dataset to find neighborhoods with the fastest response times for different categories of emergency calls. In this example, the parallel flows differ by only a few parameters.



Delta Live Tables metaprogramming with Python example

Note

This example reads sample data included in the **Databricks datasets**. Because the Databricks datasets are not supported with a pipeline that publishes to Unity Catalog, this example works only with a pipeline configured to publish to the Hive metastore. However, this pattern also works with Unity Catalog enabled pipelines, but you must read data from **external locations**. To learn more about using Unity Catalog with Delta Live Tables.

You can use a metaprogramming pattern to reduce the overhead of generating and maintaining redundant flow definitions. Metaprogramming in Delta Live Tables is done using Python inner functions. Because these functions are lazily evaluated, you can use them to create flows that are identical except for input parameters. Each invocation can include a different set of parameters that controls how each table should be generated, as shown in the following example.

Important

Because Python functions with Delta Live Tables decorators are invoked lazily, when creating datasets in a loop you must call a separate function to create the datasets to ensure correct parameter values are used. Failing to create datasets in a separate function results in multiple tables that use the parameters from the final execution of the loop.

The following example calls the create_table() function inside a loop to create tables t1 and t2:

PythonCopy

```
def create_table(name):
  @dlt.table(name=name)
  def t():
    return spark.read.table(name)


tables = ["t1", "t2"]
for t in tables:
  create_table(t)
```

PythonCopy

```
import dlt
from pyspark.sql.functions import *


@dlt.table(
  name="raw_fire_department",
  comment="raw table for fire department response"
)
@dlt.expect_or_drop("valid_received", "received IS NOT NULL")
@dlt.expect_or_drop("valid_response", "responded IS NOT NULL")
@dlt.expect_or_drop("valid_neighborhood", "neighborhood != 'None'")
def get_raw_fire_department():
  return (
    spark.read.format('csv')
      .option('header', 'true')
      .option('multiline', 'true')
```

```
   .load('/databricks-
datasets/timeseries/Fires/Fire_Department_Calls_for_Service.csv')
   .withColumnRenamed('Call Type', 'call_type')
   .withColumnRenamed('Received DtTm', 'received')
   .withColumnRenamed('Response DtTm', 'responded')
   .withColumnRenamed('Neighborhooods - Analysis Boundaries', 'neighborhood')
  .select('call_type', 'received', 'responded', 'neighborhood')
 )

all_tables = []

def generate_tables(call_table, response_table, filter):
 @dlt.table(
  name=call_table,
  comment="top level tables by call type"
 )
 def create_call_table():
  return (
   spark.sql("""
    SELECT
     unix_timestamp(received,'M/d/yyyy h:m:s a') as ts_received,
     unix_timestamp(responded,'M/d/yyyy h:m:s a') as ts_responded,
     neighborhood
    FROM LIVE.raw_fire_department
    WHERE call_type = '{filter}'
   """.format(filter=filter))
  )
  @dlt.table(
    name=response_table,
    comment="top 10 neighborhoods with fastest response time "
  )
  def create_response_table():
    return (
      spark.sql("""
        SELECT
          neighborhood,
          AVG((ts_received - ts_responded)) as response_time
        FROM LIVE.{call_table}
        GROUP BY 1
        ORDER BY response_time
        LIMIT 10
      """.format(call_table=call_table))
    )

  all_tables.append(response_table)

generate_tables("alarms_table", "alarms_response", "Alarms")
generate_tables("fire_table", "fire_response", "Structure Fire")
generate_tables("medical_table", "medical_response", "Medical Incident")

@dlt.table(
  name="best_neighborhoods",
  comment="which neighbor appears in the best response time list the most"
)
def summary():
```

```
target_tables = [dlt.read(t) for t in all_tables]

unioned = functools.reduce(lambda x,y: x.union(y), target_tables)

return (

  unioned.groupBy(col("neighborhood"))

    .agg(count("*").alias("score"))

    .orderBy(desc("score"))

)
```

Develop Delta Live Tables pipelines

Notebook experience for Delta Live Tables development

Overview of features

When you work on a Python or SQL notebook that is the source code for an existing Delta Live Tables pipeline, you can connect the notebook directly to the pipeline. When the notebook is connected to the pipeline, the following features are available:

- Start and validate the pipeline from the notebook.
- View the pipeline’s dataflow graph and event log for the latest update in the notebook.
- View pipeline diagnostics in the notebook editor.
- View the status of the pipeline’s cluster in the notebook.
- Access the Delta Live Tables UI from the notebook.

Prerequisites

- You must have an existing Delta Live Tables pipeline with a Python or SQL notebook as source code.
- You must either be the owner of the pipeline or have the CAN_MANAGE privilege.

Limitations

- The features covered in this article are only available in Azure Databricks notebooks. Workspace files are not supported.
- The web terminal is not available when attached to a pipeline. As a result, it is not visible as a tab in the bottom panel.

Connect a notebook to a Delta Live Tables pipeline

Inside the notebook, click on the drop-down menu used to select compute. The drop-down menu shows all your Delta Live Tables pipelines with this notebook as source code. To connect the notebook to a pipeline, select it from the list.

View the pipeline’s cluster status

To easily understand the state of your pipeline’s cluster, its status is shown in the compute drop-down menu with a green color to indicate that the cluster is running.

Validate pipeline code

You can validate the pipeline to check for syntax errors in your source code without processing any data.

To validate a pipeline, do one of the following:

- In the top-right corner of the notebook, click **Validate**.
- Press Shift+Enter in any notebook cell.
- In a cell's dropdown menu, click **Validate Pipeline**.

Note

If you attempt to validate your pipeline while an existing update is already running, a dialog box displays asking if you want to terminate the existing update.

Start the pipeline

A pipeline update does the following: starts a cluster, discovers and validates all the tables and views defined, and creates or updates tables and views with the most recent data available.

To start an update of your pipeline, click the **Start** button in the top-right corner of the notebook.

If you click **Yes**, the existing update stops, and a *validate* update automatically starts.

View the status of an update

The top panel in the notebook displays whether a pipeline update is:

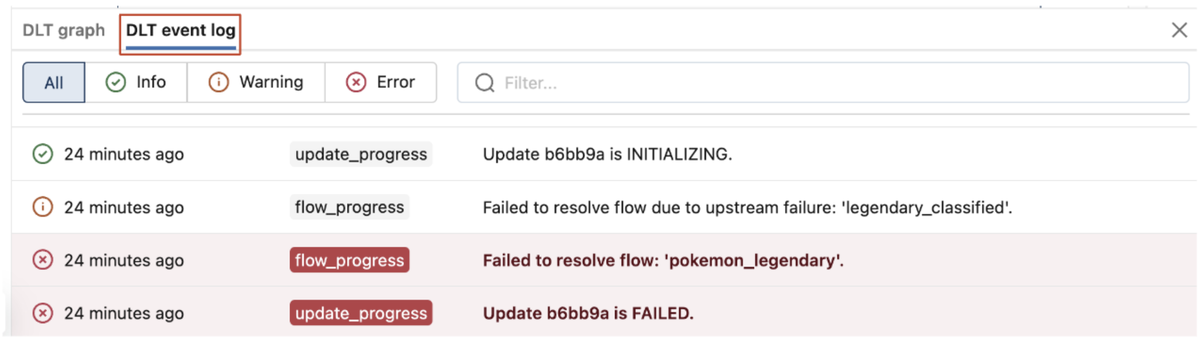
- Starting
- Validating
- Stopping

View errors and diagnostics

After a pipeline has been started or validated, any errors are shown inline with a red underline. Hover over an error to see more information.

View pipeline events

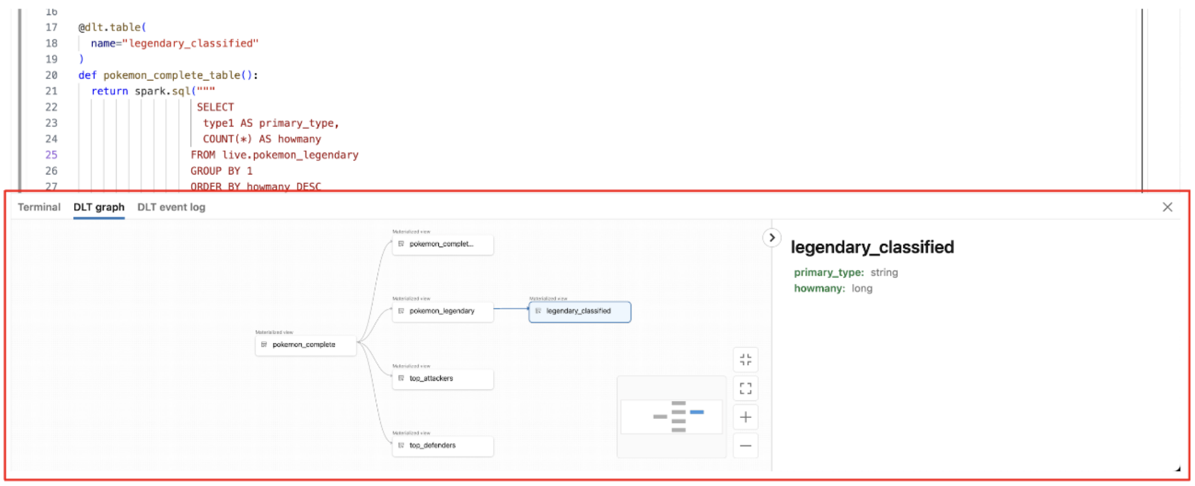
When attached to a pipeline, there is a Delta Live Tables event log tab at the bottom of the notebook.



```
24 minutes ago    update_progress    Update b6bb9a is INITIALIZING.
24 minutes ago    flow_progress      Failed to resolve flow due to upstream failure: 'legendary_classified'.
24 minutes ago    flow_progress      Failed to resolve flow: 'pokemon_legendary'.
24 minutes ago    update_progress    Update b6bb9a is FAILED.
```

View the pipeline Dataflow Graph

To view a pipeline's dataflow graph, use the Delta Live Tables graph tab at the bottom of the notebook. Selecting a node in the graph displays its schema in the right panel.

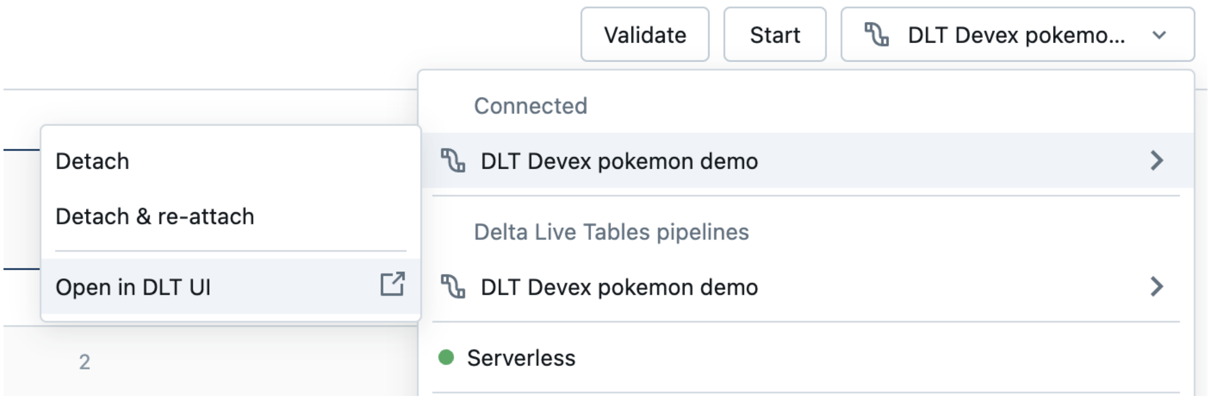


```
@dlt.table(
  name="legendary_classified"
)
def pokemon_complete_table():
  return spark.sql("""
                  SELECT
                   type1 AS primary_type,
                   COUNT(*) AS howmany
                  FROM live.pokemon_legendary
                  GROUP BY 1
                  ORDER BY howmany DESC
```

legendary_classified
primary_type: string
howmany: long

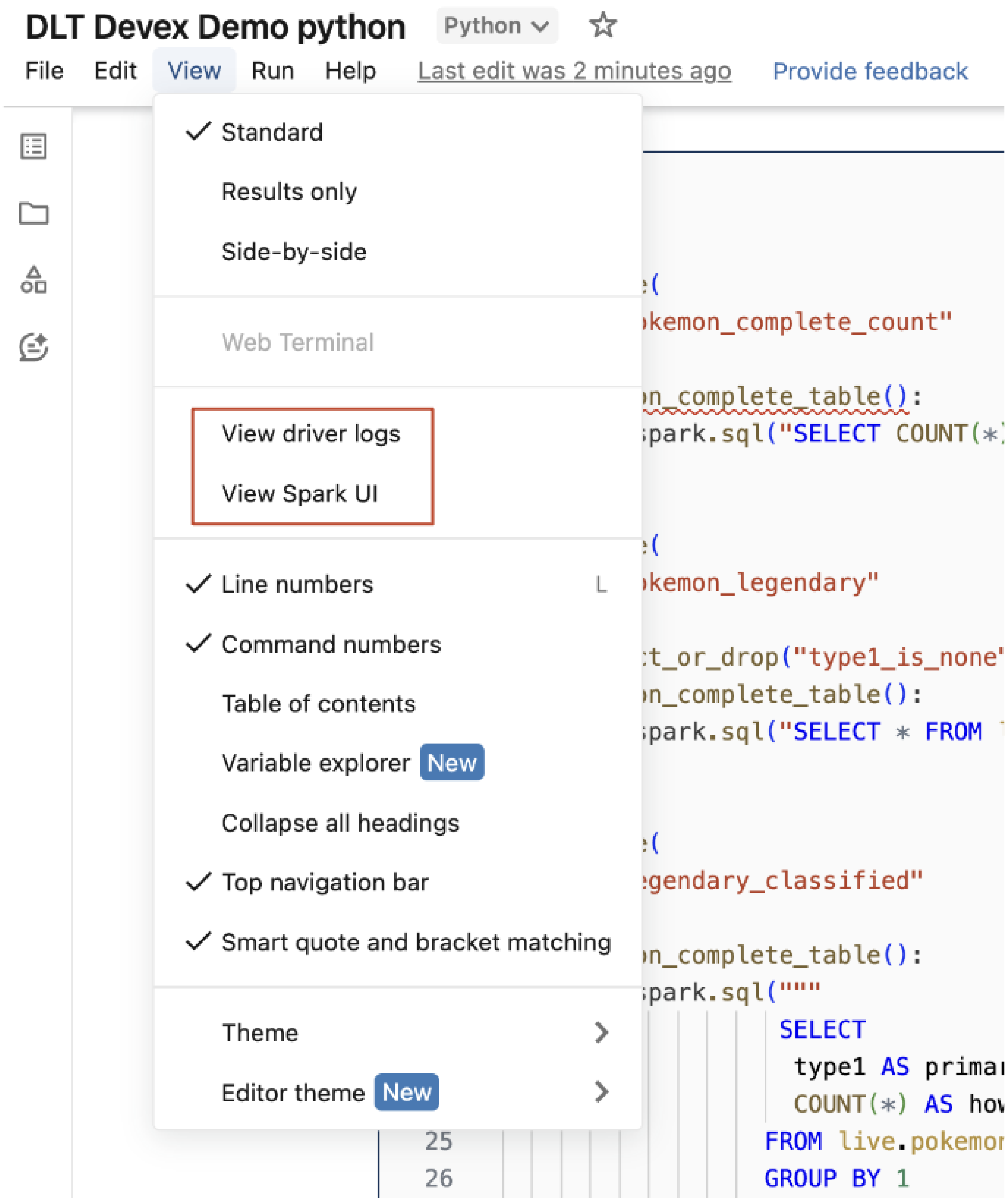
How to access the Delta Live Tables UI from the notebook

To easily jump to the Delta Live Tables UI, use the menu in the top-right corner of the notebook.



Access driver logs and the Spark UI from the notebook

The driver logs and Spark UI associated with the pipeline being developed can be easily accessed from the notebook's **View** menu.



```
:(
>kemon_complete_count"

>n_complete_table():
;park.sql("SELECT COUNT(*)

:(
>kemon_legendary"

:t_or_drop("type1_is_none"
>n_complete_table():
;park.sql("SELECT * FROM

:(
:gendary_classified"

>n_complete_table():
;park.sql("""
            SELECT
              type1 AS primai
              COUNT(*) AS hov
            FROM live.pokemor
            GROUP BY 1
```

Run an update on a Delta Live Tables pipeline

Let’s explains what a Delta Live Tables pipeline update is and how to run one.

After you create a pipeline and are ready to run it, you start an *update*. A pipeline update does the following:

- Starts a cluster with the correct configuration.
- Discovers all the tables and views defined, and checks for any analysis errors such as invalid column names, missing dependencies, and syntax errors.
- Creates or updates tables and views with the most recent data available.

You can check for problems in a pipeline’s source code without waiting for tables to be created or updated using a Validate update. The Validate feature is useful when developing or testing pipelines by allowing you to quickly find and fix errors in your pipeline, such as incorrect table or column names.

Start a pipeline update

Azure Databricks provides several options to start pipeline updates, including the following:

- In the Delta Live Tables UI, you have the following options:
    - Click the Start button on the pipeline details page.
    - From the pipelines list, click ▶ in the **Actions** column.
- To start an update in a notebook, click **Delta Live Tables > Start** in the notebook toolbar.
- You can trigger pipelines programmatically using the API or CLI.
- You can schedule the pipeline as a job using the Delta Live Tables UI or the jobs UI.

How Delta Live Tables updates tables and views

The tables and views updated, and how those tables are views are updated, depends on the update type:

- **Refresh all**: All live tables are updated to reflect the current state of their input data sources. For all streaming tables, new rows are appended to the table.
- **Full refresh all**: All live tables are updated to reflect the current state of their input data sources. For all streaming tables, Delta Live Tables attempts to clear all data from each table and then load all data from the streaming source.

- **Refresh selection**: The behavior of refresh selection is identical to refresh all, but allows you to refresh only selected tables. Selected live tables are updated to reflect the current state of their input data sources. For selected streaming tables, new rows are appended to the table.
- **Full refresh selection**: The behavior of full refresh selection is identical to full refresh all, but allows you to perform a full refresh of only selected tables. Selected live tables are updated to reflect the current state of their input data sources. For selected streaming tables, Delta Live Tables attempts to clear all data from each table and then load all data from the streaming source.

For existing live tables, an update has the same behavior as a SQL REFRESH on a materialized view. For new live tables, the behavior is the same as a SQL CREATE operation.

Start a pipeline update for selected tables

You may want to reprocess data for only selected tables in your pipeline. For example, during development, you only change a single table and want to reduce testing time, or a pipeline update fails and you want to refresh only the failed tables.

Note

You can use selective refresh with only triggered pipelines.

To start an update that refreshes selected tables only, on the **Pipeline details** page:

1. Click **Select tables for refresh**. The **Select tables for refresh** dialog appears.

If you do not see the **Select tables for refresh** button, make sure the **Pipeline details** page displays the latest update, and the update is complete. If a DAG is not displayed for the latest update, for example, because the update failed, the **Select tables for refresh** button is not displayed.

2. To select the tables to refresh, click on each table. The selected tables are highlighted and labeled. To remove a table from the update, click on the table again.
3. Click **Refresh selection**.

Note

The **Refresh selection** button displays the number of selected tables in parentheses.

To reprocess data that has already been ingested for the selected tables, click ⌄ next to the **Refresh selection** button and click **Full Refresh selection**.

Start a pipeline update for failed tables

If a pipeline update fails because of errors in one or more tables in the pipeline graph, you can start an update of only failed tables and any downstream dependencies.

Note

Excluded tables are not refreshed, even if they depend on a failed table.

To update failed tables, on the **Pipeline details** page, click **Refresh failed tables**.

To update only selected failed tables:

1. Click next to the **Refresh failed tables** button and click **Select tables for refresh**. The **Select tables for refresh** dialog appears.
2. To select the tables to refresh, click on each table. The selected tables are highlighted and labeled. To remove a table from the update, click on the table again.
3. Click **Refresh selection**.

Note

The **Refresh selection** button displays the number of selected tables in parentheses.

To reprocess data that has already been ingested for the selected tables, click next to the **Refresh selection** button and click **Full Refresh selection**.

Check a pipeline for errors without waiting for tables to update

Important

The Delta Live Tables Validate update feature is in Public Preview.

To check whether a pipeline's source code is valid without running a full update, use *Validate*. A Validate update resolves the definitions of datasets and flows defined in the pipeline but does not materialize or publish any datasets. Errors found during validation, such as incorrect table or column names, are reported in the UI.

To run a Validate update, on the pipeline details page click next to **Start** and click **Validate**.

After the Validate update completes, the event log shows events related only to the Validate update, and no metrics are displayed in the DAG. If errors are found, details are available in the event log.

You can see results for only the most recent Validate update. If the Validate update was the most recently run update, you can see the results by selecting it in the update history. If another update is run after the Validate update, the results are no longer available in the UI.

Continuous vs. triggered pipeline execution

If the pipeline uses the **triggered** execution mode, the system stops processing after successfully refreshing all tables or selected tables in the pipeline once, ensuring each table that is part of the update is updated based on the data available when the update started.

If the pipeline uses **continuous** execution, Delta Live Tables processes new data as it arrives in data sources to keep tables throughout the pipeline fresh.

The execution mode is independent of the type of table being computed. Both materialized views and streaming tables can be updated in either execution mode. To avoid unnecessary processing in continuous execution mode, pipelines automatically monitor dependent Delta tables and perform an update only when the contents of those dependent tables have changed.

Note

The Delta Live Tables runtime is not able to detect changes in non-Delta data sources. The table is still updated regularly, but with a higher default trigger interval to prevent excessive recomputation from slowing down any incremental processing happening on the cluster.

Table comparing data pipeline execution modes

The following table highlights differences between these execution modes:

Expand table

	Triggered	**Continuous**
When does the update stop?	Automatically once complete.	Runs continuously until manually stopped.
What data is processed?	Data available when the update is started.	All data as it arrives at configured sources.
What data freshness requirements is this best for?	Data updates run every 10 minutes, hourly, or daily.	Data updates desired between every 10 seconds and a few minutes.

Triggered pipelines can reduce resource consumption and expense since the cluster runs only long enough to execute the pipeline. However, new data won't be processed until the pipeline is triggered. Continuous pipelines require an always-running cluster, which is more expensive but reduces processing latency.

You can configure execution mode with the **Pipeline mode** option in the settings.

How to choose pipeline boundaries

A Delta Live Tables pipeline can process updates to a single table, many tables with dependent relationship, many tables without relationships, or multiple independent flows of tables with dependent relationships. This section contains considerations to help determine how to break up your pipelines.

Larger Delta Live Tables pipelines have a number of benefits. These include the following:

- More efficiently use cluster resources.
- Reduce the number of pipelines in your workspace.
- Reduce the complexity of workflow orchestration.

Some common recommendations on how processing pipelines should be split include the following:

- Split functionality at team boundaries. For example, your data team may maintain pipelines to transform data while your data analysts maintain pipelines that analyze the transformed data.
- Split functionality at application-specific boundaries to reduce coupling and facilitate the re-use of common functionality.

Development and production modes

You can optimize pipeline execution by switching between development and production modes. Use the

buttons in the Pipelines UI to switch between these two modes. By default, pipelines run in development mode.

When you run your pipeline in development mode, the Delta Live Tables system does the following:

- Reuses a cluster to avoid the overhead of restarts. By default, clusters run for two hours when development mode is enabled. You can change this with the pipelines.clusterShutdown.delay setting in the Configure your compute settings.
- Disables pipeline retries so you can immediately detect and fix errors.

In production mode, the Delta Live Tables system does the following:

- Restarts the cluster for specific recoverable errors, including memory leaks and stale credentials.

- Retries execution in the event of specific errors, for example, a failure to start a cluster.

Note

Switching between development and production modes only controls cluster and pipeline execution behavior. Storage locations and target schemas in the catalog for publishing tables must be configured as part of pipeline settings and are not affected when switching between modes.

Schedule a pipeline

You can start a triggered pipeline manually or run the pipeline on a schedule with an Azure Databricks job. You can create and schedule a job with a single pipeline task directly in the Delta Live Tables UI or add a pipeline task to a multi-task workflow in the jobs UI.

To create a single-task job and a schedule for the job in the Delta Live Tables UI:

1. Click **Schedule > Add a schedule**. The **Schedule** button is updated to show the number of existing schedules if the pipeline is included in one or more scheduled jobs, for example, **Schedule (5)**.
2. Enter a name for the job in the **Job name** field.
3. Set the **Schedule** to **Scheduled**.
4. Specify the period, starting time, and time zone.
5. Configure one or more email addresses to receive alerts on pipeline start, success, or failure.
6. Click **Create**.

Run a Delta Live Tables pipeline in a workflow

You can run a Delta Live Tables pipeline as part of a data processing workflow with Databricks jobs, Apache Airflow, or Azure Data Factory.

Jobs

You can orchestrate multiple tasks in a Databricks job to implement a data processing workflow. To include a Delta Live Tables pipeline in a job, use the **Pipeline** task when you create a job.

Apache Airflow

Apache Airflow is an open source solution for managing and scheduling data workflows. Airflow represents workflows as directed acyclic graphs (DAGs) of operations. You define a workflow in a Python file and Airflow manages the scheduling and execution.

To run a Delta Live Tables pipeline as part of an Airflow workflow, use the DatabricksSubmitRunOperator.

Requirements

The following are required to use the Airflow support for Delta Live Tables:

- Airflow version 2.1.0 or later.
- The Databricks provider package version 2.1.0 or later.

Example

The following example creates an Airflow DAG that triggers an update for the Delta Live Tables pipeline with the identifier 8279d543-063c-4d63-9926-dae38e35ce8b:

PythonCopy

```
from airflow import DAG

from airflow.providers.databricks.operators.databricks import DatabricksSubmitRunOperator

from airflow.utils.dates import days_ago


default_args = {

  'owner': 'airflow'

}
```

```
with DAG('dlt',
        start_date=days_ago(2),
        schedule_interval="@once",
        default_args=default_args
        ) as dag:


  opr_run_now=DatabricksSubmitRunOperator(
    task_id='run_now',
    databricks_conn_id='CONNECTION_ID',
    pipeline_task={"pipeline_id": "8279d543-063c-4d63-9926-dae38e35ce8b"}
  )
```

Replace CONNECTION_ID with the identifier for an Airflow connection to your workspace.

Save this example in the airflow/dags directory and use the Airflow UI to view and trigger the DAG. Use the Delta Live Tables UI to view the details of the pipeline update.

Azure Data Factory

Azure Data Factory is a cloud-based ETL service that lets you orchestrate data integration and transformation workflows. Azure Data Factory directly supports running Azure Databricks tasks in a workflow, including notebooks, JAR tasks, and Python scripts. You can also include a pipeline in a workflow by calling the Delta Live Tables API from an Azure Data Factory Web activity. For example, to trigger a pipeline update from Azure Data Factory:

1. Create a data factory or open an existing data factory.
2. When creation completes, open the page for your data factory and click the **Open Azure Data Factory Studio** tile. The Azure Data Factory user interface appears.
3. Create a new Azure Data Factory pipeline by selecting **Pipeline** from the **New** drop-down menu in the Azure Data Factory Studio user interface.
4. In the **Activities** toolbox, expand **General** and drag the **Web** activity to the pipeline canvas. Click the **Settings** tab and enter the following values:

Note

As a security best practice, when you authenticate with automated tools, systems, scripts, and apps, Databricks recommends that you use personal access tokens belonging to **service principals** instead of workspace users. To create tokens for service principals.

- **URL**: https://<databricks-instance>/api/2.0/pipelines/<pipeline-id>/updates.

Replace <get-workspace-instance>.

Replace <pipeline-id> with the pipeline identifier.

- **Method**: Select **POST** from the drop-down menu.
- **Headers**: Click **+ New**. In the **Name** text box, enter Authorization. In the **Value** text box, enter Bearer <personal-access-token>.

Replace <personal-access-token> with an Azure Databricks personal access token.

- **Body**: To pass additional request parameters, enter a JSON document containing the parameters. For example, to start an update and reprocess all data for the pipeline: {"full_refresh": "true"}. If there are no additional request parameters, enter empty braces ({}).

To test the Web activity, click **Debug** on the pipeline toolbar in the Data Factory UI. The output and status of the run, including errors, are displayed in the **Output** tab of the Azure Data Factory pipeline. Use the Delta Live Tables UI to view the details of the pipeline update.

Tip

A common workflow requirement is to start a task after completion of a previous task. Because the Delta Live Tables updates request is asynchronous—the request returns after starting the update but before the update completes—tasks in your Azure Data Factory pipeline with a dependency on the Delta Live Tables update must wait for the update to complete. An option to wait for update completion is adding an **Until activity** following the Web activity that triggers the Delta Live Tables update. In the Until activity:

1. Add a **Wait activity** to wait a configured number of seconds for update completion.
2. Add a Web activity following the Wait activity that uses the Delta Live Tables **Get update details** request to get the status of the update. The state field in the response returns the current state of the update, including if it has completed.

3. Use the value of the state field to set the terminating condition for the Until activity. You can also use a **Set Variable activity** to add a pipeline variable based on the state value and use this variable for the terminating condition.

Publish data from Delta Live Tables pipelines to the Hive metastore

You can make the output data of your pipeline discoverable and available to query by publishing datasets to the Hive metastore. To publish datasets to the metastore, enter a schema name in the **Target** field when you create a pipeline. You can also add a target database to an existing pipeline.

By default, all tables and views created in Delta Live Tables are local to the pipeline. You must publish tables to a target schema to query or use Delta Live Tables datasets outside the pipeline in which they are declared.

How to publish Delta Live Tables datasets to a schema

You can declare a target schema for all tables in your Delta Live Tables pipeline using the **Target schema** field in the **Pipeline settings** and **Create pipeline** UIs.

You can also specify a schema in a JSON configuration by setting the target value.

You must run an update for the pipeline to publish results to the target schema.

You can use this feature with multiple environment configurations to publish to different schemas based on the environment. For example, you can publish to a dev schema for development and a prod schema for production data.

How to query datasets in Delta Live Tables

After an update completes, you can view the schema and tables, query the data, or use the data in downstream applications.

Once published, Delta Live Tables tables can be queried from any environment with access to the target schema. This includes Databricks SQL, notebooks, and other Delta Live Tables pipelines.

Important

When you create a target configuration, only tables and associated metadata are published. Views are not published to the metastore.

Exclude tables from target schema

If you need to calculate intermediate tables that are not intended for external consumption, you can prevent them from being published to a schema using the TEMPORARY keyword. Temporary tables still store and process data according to Delta Live Tables semantics, but should not be accessed outside of the current pipeline. A temporary table persists for the lifetime of the pipeline that creates it. Use the following syntax to declare temporary tables:

SQL

```
CREATE TEMPORARY LIVE TABLE temp_table
AS SELECT … ;
```

Python

```
@dlt.table(
  temporary=True)
def temp_table():
  return ("...")
```

Use Unity Catalog with your Delta Live Tables pipelines

In addition to the existing support for persisting tables to the Hive metastore, you can use Unity Catalog with your Delta Live Tables pipelines to:

- Define a catalog in Unity Catalog where your pipeline will persist tables.
- Read data from Unity Catalog tables.

Your workspace can contain pipelines that use Unity Catalog or the Hive metastore. However, a single pipeline cannot write to both the Hive metastore and Unity Catalog and existing pipelines cannot be upgraded to use Unity Catalog. Your existing pipelines that do not use Unity Catalog are not affected by this preview, and will continue to persist data to the Hive metastore using the configured storage location.

Unless specified otherwise in this document, all existing data sources and Delta Live Tables functionality are supported with pipelines that use Unity Catalog. Both the Python and SQL interfaces are supported with pipelines that use Unity Catalog.

The tables created in your pipeline can also be queried from shared Unity Catalog clusters using Databricks Runtime 13.3 LTS and above or a SQL warehouse. Tables cannot be queried from assigned or no isolation clusters.

To manage permissions on the tables created by a Unity Catalog pipeline, use GRANT and REVOKE.

Requirements

The following are required to create tables in Unity Catalog from a Delta Live Tables pipeline:

- You must have USE CATALOG privileges on the target catalog.
- You must have CREATE MATERIALIZED VIEW and USE SCHEMA privileges in the target schema if your pipeline creates materialized views.
- You must have CREATE TABLE and USE SCHEMA privileges in the target schema if your pipeline creates streaming tables.
- If a target schema is not specified in the pipeline settings, you must have CREATE MATERIALIZED VIEW or CREATE TABLE privileges on at least one schema in the target catalog.

Limitations

The following are limitations when using Unity Catalog with Delta Live Tables:

- By default, only the pipeline owner and workspace admins have permission to view the driver logs from the cluster that runs a Unity Catalog-enabled pipeline. To enable access for other users to view the driver logs.
- Existing pipelines that use the Hive metastore cannot be upgraded to use Unity Catalog. To migrate an existing pipeline that writes to Hive metastore, you must create a new pipeline and re-ingest data from the data source(s).
- You cannot create a Unity Catalog-enabled pipeline in a workspace attached to a metastore created during the Unity Catalog public preview.
- Third-party libraries and JARs are not supported.
- Data manipulation language (DML) queries that modify the schema of a streaming table are not supported.
- A materialized view created in a Delta Live Tables pipeline cannot be used as a streaming source outside of that pipeline, for example, in another pipeline or in a downstream notebook.
- Publishing to schemas that specify a managed storage location is supported only in the preview channel.
- If a pipeline publishes to a schema with a managed storage location, the schema can be changed in a subsequent update, but only if the updated schema uses the same storage location as the previously specified schema.
- If the target schema specifies a storage location, all tables are stored there. If a schema storage location is not specified, tables are stored in the catalog storage location if the target catalog specifies one. If schema and catalog storage locations are not specified, tables are stored in the root storage location of the metastore where the tables are published.
- The **History** tab in Catalog Explorer does not show history for streaming tables and materialized views.
- The LOCATION property is not supported when defining a table.
- Unity Catalog-enabled pipelines cannot publish to the Hive metastore.
- Python UDF support is in Public Preview. To use Python UDFs, your pipeline must use the preview channel.
- You cannot use Delta Sharing with a Delta Live Tables materialized view or streaming table published to Unity Catalog.
- You cannot use the event_log table valued function in a pipeline or query to access the event logs of multiple pipelines.

- You cannot share a view created over the event_log table valued function with other users.
- Single-node clusters are not supported with Unity Catalog-enabled pipelines. Because Delta Live Tables might create a single-node cluster to run smaller pipelines, your pipeline might fail with an error message referencing single-node mode. If this occurs, make sure you specify at least one worker when you Configure your compute settings.
- Tables created in a Unity Catalog-enabled pipeline cannot be queried from assigned or no isolation clusters. To query tables created by a Delta Live Tables pipeline, you must use a shared access mode cluster using Databricks Runtime 13.3 LTS and above or a SQL warehouse.
- Delta Live Tables uses a shared access mode cluster to run a Unity Catalog-enabled pipeline. A Unity Catalog-enabled pipeline cannot run on an assigned cluster.
- You cannot use row filters or column masks with materialized views or streaming tables published to Unity Catalog.

Note

The underlying files supporting materialized views might include data from upstream tables (including possible personally identifiable information) that do not appear in the materialized view definition. This data is automatically added to the underlying storage to support incremental refreshing of materialized views.

Because the underlying files of a materialized view might risk exposing data from upstream tables not part of the materialized view schema, Databricks recommends not sharing the underlying storage with untrusted downstream consumers.

For example, suppose the definition of a materialized view includes a COUNT(DISTINCT field_a) clause. Even though the materialized view definition only includes the aggregate COUNT DISTINCT clause, the underlying files will contain a list of the actual values of field_a.

Changes to existing functionality

When Delta Live Tables is configured to persist data to Unity Catalog, the lifecycle of the table is managed by the Delta Live Tables pipeline. Because the pipeline manages the table lifecycle and permissions:

- When a table is removed from the Delta Live Tables pipeline definition, the corresponding materialized view or streaming table entry is removed from Unity Catalog on the next pipeline update. The actual data is retained for a period of time so that it can be recovered if it was deleted by mistake. The data can be recovered by adding the materialized view or streaming table back into the pipeline definition.

- Deleting the Delta Live Tables pipeline results in deletion of all tables defined in that pipeline. Because of this change, the Delta Live Tables UI is updated to prompt you to confirm deletion of a pipeline.
- Internal backing tables, including backing tables used to support APPLY CHANGES INTO, are not directly accessible by users.

Write tables to Unity Catalog from a Delta Live Tables pipeline

To write your tables to Unity Catalog, when you create a pipeline, select **Unity Catalog** under **Storage options**, select a catalog in the **Catalog** drop-down menu, and select a database name in the **Target schema** drop-down menu.

Ingest data into a Unity Catalog pipeline

Your pipeline configured to use Unity Catalog can read data from:

- Unity Catalog managed and external tables, views, materialized views and streaming tables.
- Hive metastore tables and views.
- Auto Loader using the cloud_files() function to read from Unity Catalog external locations.
- Apache Kafka and Amazon Kinesis.

The following are examples of reading from Unity Catalog and Hive metastore tables.

Batch ingestion from a Unity Catalog table

SQL

```
CREATE OR REFRESH LIVE TABLE
  table_name
AS SELECT
  *
FROM
  my_catalog.my_schema.table1;
```

Python

```
@dlt.table
```

```
def table_name():
  return spark.table("my_catalog.my_schema.table")
```

Stream changes from a Unity Catalog table

SQL

```
CREATE OR REFRESH STREAMING TABLE
  table_name
AS SELECT
  *
FROM
  STREAM(my_catalog.my_schema.table1);
```

Python

```
@dlt.table
def table_name():
  return spark.readStream.table("my_catalog.my_schema.table")
```

Ingest data from Hive metastore

A pipeline that uses Unity Catalog can read data from Hive metastore tables using the hive_metastore catalog:

SQL

```
CREATE OR REFRESH LIVE TABLE
  table_name
AS SELECT
  *
FROM
  hive_metastore.some_schema.table;
```

Python

```
@dlt.table
def table3():
  return spark.table("hive_metastore.some_schema.table")
```

Ingest data from Auto Loader

SQL

```
CREATE OR REFRESH STREAMING TABLE
  table_name
AS SELECT
  *
FROM
  cloud_files(
    <path-to-uc-external-location>,
    "json"
  )
```

Python

```
@dlt.table(table_properties={"quality": "bronze"})
def table_name():
  return (
     spark.readStream.format("cloudFiles")
     .option("cloudFiles.format", "json")
     .load(f"{path_to_uc_external_location}")
 )
```

Share materialized views (live tables)

By default, the tables created by a pipeline can be queried only by the pipeline owner. You can give other users the ability to query a table by using GRANT statements and you can revoke query access using REVOKE statements.

Grant select on a table

SQL

```
GRANT SELECT ON TABLE
  my_catalog.my_schema.live_table
TO
  `user@databricks.com`
```

Revoke select on a table

SQL

```
REVOKE SELECT ON TABLE
  my_catalog.my_schema.live_table
FROM
  `user@databricks.com`
```

Grant create table or create materialized view privileges

SQL

```
GRANT CREATE { MATERIALIZED VIEW | TABLE } ON SCHEMA
  my_catalog.my_schema
TO
  { principal | user }
```

View lineage for a pipeline

Lineage for tables in a Delta Live Tables pipeline is visible in Catalog Explorer. For materialized views or streaming tables in a Unity Catalog-enabled pipeline, the Catalog Explorer lineage UI shows the upstream and downstream tables.

For a materialized view or streaming table in a Unity Catalog-enabled Delta Live Tables pipeline, the Catalog Explorer lineage UI will also link to the pipeline that produced the materialized view or streaming table if the pipeline is accessible from the current workspace.

Add, change, or delete data in a streaming table

You can use data manipulation language (DML) statements, including insert, update, delete, and merge statements, to modify streaming tables published to Unity Catalog. Support for DML queries against streaming tables enables use cases such as updating tables for General Data Protection Regulation (GDPR) compliance.

Note

- DML statements that modify the table schema of a streaming table are not supported. Ensure that your DML statements do not attempt to evolve the table schema.
- DML statements that update a streaming table can be run only in a shared Unity Catalog cluster or a SQL warehouse using Databricks Runtime 13.3 LTS and above.

- Because streaming requires append-only data sources, if your processing requires streaming from a source streaming table with changes (for example, by DML statements), set the **skipChangeCommits flag** when reading the source streaming table. When skipChangeCommits is set, transactions that delete or modify records on the source table are ignored. If your processing does not require a streaming table, you can use a materialized view (which does not have the append-only restriction) as the target table.

The following are examples of DML statements to modify records in a streaming table.

Delete records with a specific ID:

SQL

```
DELETE FROM my_streaming_table WHERE id = 123;
```

Update records with a specific ID:

SQL

```
UPDATE my_streaming_table SET name = 'Jane Doe' WHERE id = 123;
```

Load data with Delta Live Tables

You can load data from any data source supported by Apache Spark on Azure Databricks using Delta Live Tables. You can define datasets (tables and views) in Delta Live Tables against any query that returns a Spark DataFrame, including streaming DataFrames and Pandas for Spark DataFrames. For data ingestion tasks, Databricks recommends using streaming tables for most use cases. Streaming tables are good for ingesting data from cloud object storage using Auto Loader or from message buses like Kafka. The examples below demonstrate some common patterns.

Important

Not all data sources have SQL support. You can mix SQL and Python notebooks in a Delta Live Tables pipeline to use SQL for all operations beyond ingestion.

Load files from cloud object storage

Databricks recommends using Auto Loader with Delta Live Tables for most data ingestion tasks from cloud object storage. Auto Loader and Delta Live Tables are designed to incrementally and idempotently load ever-growing data as it arrives in cloud storage. The following examples use Auto Loader to create datasets from CSV and JSON files:

Note

To load files with Auto Loader in a Unity Catalog enabled pipeline, you must use **external locations**.

Python

```
@dlt.table

def customers():

  return (

    spark.readStream.format("cloudFiles")

      .option("cloudFiles.format", "csv")

      .load("/databricks-datasets/retail-org/customers/")

  )


@dlt.table

def sales_orders_raw():

  return (
    spark.readStream.format("cloudFiles")
      .option("cloudFiles.format", "json")
      .load("/databricks-datasets/retail-org/sales_orders/")
  )
```

SQL

```
CREATE OR REFRESH STREAMING TABLE customers
AS SELECT * FROM cloud_files("/databricks-datasets/retail-org/customers/", "csv")


CREATE OR REFRESH STREAMING TABLE sales_orders_raw
AS SELECT * FROM cloud_files("/databricks-datasets/retail-org/sales_orders/", "json")
```

Warning

If you use Auto Loader with file notifications and run a full refresh for your pipeline or streaming table, you must manually clean up your resources. You can use the **CloudFilesResourceManager** in a notebook to perform cleanup.

Load data from a message bus

You can configure Delta Live Tables pipelines to ingest data from message buses with streaming tables. Databricks recommends combining streaming tables with continuous execution and enhanced autoscaling to provide the most efficient ingestion for low-latency loading from message buses.

For example, the following code configures a streaming table to ingest data from Kafka:

Python

```
import dlt


@dlt.table
def kafka_raw():
  return (
    spark.readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", "<server:ip>")
      .option("subscribe", "topic1")

      .option("startingOffsets", "latest")

      .load()

  )
```

You can write downstream operations in pure SQL to perform streaming transformations on this data, as in the following example:

SQL

```
CREATE OR REFRESH STREAMING TABLE streaming_silver_table

AS SELECT

  *

FROM

  STREAM(LIVE.kafka_raw)

WHERE ...
```

Load data from external systems

Delta Live Tables supports loading data from any data source supported by Azure Databricks. You can also load external data using Lakehouse Federation for supported data sources. Because Lakehouse Federation requires Databricks Runtime 13.3 LTS or above, to use Lakehouse Federation your pipeline must be configured to use the preview channel.

Some data sources do not have equivalent support in SQL. If you cannot use Lakehouse Federation with one of these data sources, you can use a standalone Python notebook to ingest data from the source. This notebook can then be added as a source library with SQL notebooks to build a Delta Live Tables pipeline. The following example declares a materialized view to access the current state of data in a remote PostgreSQL table:

Python

```
import dlt


@dlt.table

def postgres_raw():

  return (
  spark.read
    .format("postgresql")
    .option("dbtable", table_name)
    .option("host", database_host_url)
    .option("port", 5432)
    .option("database", database_name)
    .option("user", username)
    .option("password", password)
    .load()
 )
```

Load small or static datasets from cloud object storage

You can load small or static datasets using Apache Spark load syntax. Delta Live Tables supports all of the file formats supported by Apache Spark on Azure Databricks.

The following examples demonstrate loading JSON to create Delta Live Tables tables:

Python

```
@dlt.table
def clickstream_raw():
  return (spark.read.format("json").load("/databricks-datasets/wikipedia-datasets/data-001/clickstream/raw-uncompressed-json/2015_2_clickstream.json"))
```

SQL

```
CREATE OR REFRESH LIVE TABLE clickstream_raw
AS SELECT * FROM json.`/databricks-datasets/wikipedia-datasets/data-001/clickstream/raw-uncompressed-json/2015_2_clickstream.json`;
```

Note

The SELECT * FROM format.`path`; SQL construct is common to all SQL environments on Azure Databricks. It is the recommended pattern for direct file access using SQL with Delta Live Tables.

Securely access storage credentials with secrets in a pipeline

You can use Azure Databricks secrets to store credentials such as access keys or passwords. To configure the secret in your pipeline, use a Spark property in the pipeline settings cluster configuration.

The following example uses a secret to store an access key required to read input data from an Azure Data Lake Storage Gen2 (ADLS Gen2) storage account using Auto Loader. You can use this same method to configure any secret required by your pipeline, for example, AWS keys to access S3, or the password to an Apache Hive metastore.

Note

You must add the spark.hadoop. prefix to the spark_conf configuration key that sets the secret value.

JSON

```
{
  "id": "43246596-a63f-11ec-b909-0242ac120002",
  "clusters": [
   {
     "spark_conf": {
      "spark.hadoop.fs.azure.account.key.<storage-account-
name>.dfs.core.windows.net": "{{secrets/<scope-name>/<secret-name>}}"
     },
     "autoscale": {
      "min_workers": 1,
      "max_workers": 5,
      "mode": "ENHANCED"
     }
   }
  ],
  "development": true,
  "continuous": false,
  "libraries": [
   {
      "notebook": {
        "path": "/Users/user@databricks.com/DLT Notebooks/Delta Live Tables quickstart"
      }
    }
  ],
  "name": "DLT quickstart using ADLS2"
}
```

Replace

- <storage-account-name> with the ADLS Gen2 storage account name.
- <scope-name> with the Azure Databricks secret scope name.
- <secret-name> with the name of the key containing the Azure storage account access key.

Python

```python
import dlt


json_path = "abfss://<container-name>@<storage-account-name>.dfs.core.windows.net/<path-to-input-dataset>"
@dlt.create_table(
 comment="Data ingested from an ADLS2 storage account."
)
def read_from_ADLS2():
 return (
  spark.readStream.format("cloudFiles")
   .option("cloudFiles.format", "json")
   .load(json_path)
 )
```

Replace

- <container-name> with the name of the Azure storage account container that stores the input data.

- <storage-account-name> with the ADLS Gen2 storage account name.
- <path-to-input-dataset> with the path to the input dataset.

Transform data with Delta Live Tables

how you can use Delta Live Tables to declare transformations on datasets and specify how records are processed through query logic. It also contains some examples of common transformation patterns that can be useful when building out Delta Live Tables pipelines.

You can define a dataset against any query that returns a DataFrame. You can use Apache Spark built-in operations, UDFs, custom logic, and MLflow models as transformations in your Delta Live Tables pipeline. Once data has been ingested into your Delta Live Tables pipeline, you can define new datasets against upstream sources to create new streaming tables, materialized views, and views.

When to use views, materialized views, and streaming tables

To ensure your pipelines are efficient and maintainable, choose the best dataset type when you implement your pipeline queries.

Consider using a view when:

- You have a large or complex query that you want to break into easier-to-manage queries.
- You want to validate intermediate results using expectations.
- You want to reduce storage and compute costs and do not require the materialization of query results. Because tables are materialized, they require additional computation and storage resources.

Consider using a materialized view when:

- Multiple downstream queries consume the table. Because views are computed on demand, the view is re-computed every time the view is queried.
- Other pipelines, jobs, or queries consume the table. Because views are not materialized, you can only use them in the same pipeline.
- You want to view the results of a query during development. Because tables are materialized and can be viewed and queried outside of the pipeline, using tables during development can help validate the correctness of computations. After validating, convert queries that do not require materialization into views.

Consider using a streaming table when:

- A query is defined against a data source that is continuously or incrementally growing.
- Query results should be computed incrementally.

- High throughput and low latency is desired for the pipeline.

Note

Streaming tables are always defined against streaming sources. You can also use streaming sources with APPLY CHANGES INTO to apply updates from CDC feeds.

Combine streaming tables and materialized views in a single pipeline

Streaming tables inherit the processing guarantees of Apache Spark Structured Streaming and are configured to process queries from append-only data sources, where new rows are always inserted into the source table rather than modified.

Note

Although, by default, streaming tables require append-only data sources, when a streaming source is another streaming table that requires updates or deletes, you can override this behavior with the **skipChangeCommits flag**.

A common streaming pattern includes ingesting source data to create the initial datasets in a pipeline. These initial datasets are commonly called *bronze* tables and often perform simple transformations.

By contrast, the final tables in a pipeline, commonly referred to as *gold* tables, often require complicated aggregations or reading from sources that are the targets of an APPLY CHANGES INTO operation. Because these operations inherently create updates rather than appends, they are not supported as inputs to streaming tables. These transformations are better suited for materialized views.

By mixing streaming tables and materialized views into a single pipeline, you can simplify your pipeline, avoid costly re-ingestion or re-processing of raw data, and have the full power of SQL to compute complex aggregations over an efficiently encoded and filtered dataset. The following example illustrates this type of mixed processing:

Note

These examples use Auto Loader to load files from cloud storage. To load files with Auto Loader in a Unity Catalog enabled pipeline, you must use **external locations**.

Python

```
@dlt.table

def streaming_bronze():

 return (

  # Since this is a streaming source, this table is incremental.

  spark.readStream.format("cloudFiles")
```

```
    .option("cloudFiles.format", "json")
    .load("abfss://path/to/raw/data")
 )

@dlt.table
def streaming_silver():
 # Since we read the bronze table as a stream, this silver table is also
 # updated incrementally.
 return dlt.read_stream("streaming_bronze").where(...)

@dlt.table
def live_gold():
 # This table will be recomputed completely by reading the whole silver table
 # when it is updated.
 return dlt.read("streaming_silver").groupBy("user_id").count()
```

SQL

```
CREATE OR REFRESH STREAMING TABLE streaming_bronze
AS SELECT * FROM cloud_files(
 "abfss://path/to/raw/data", "json"
)

CREATE OR REFRESH STREAMING TABLE streaming_silver
AS SELECT * FROM STREAM(LIVE.streaming_bronze) WHERE...

CREATE OR REFRESH LIVE TABLE live_gold
AS SELECT count(*) FROM LIVE.streaming_silver GROUP BY user_id
```

Learn more about using Auto Loader to efficiently read JSON files from Azure storage for incremental processing.

Stream-static joins

Stream-static joins are a good choice when denormalizing a continuous stream of append-only data with a primarily static dimension table.

With each pipeline update, new records from the stream are joined with the most current snapshot of the static table. If records are added or updated in the static table after corresponding data from the streaming table has been processed, the resultant records are not recalculated unless a full refresh is performed.

In pipelines configured for triggered execution, the static table returns results as of the time the update started. In pipelines configured for continuous execution, each time the table processes an update, the most recent version of the static table is queried.

The following is an example of a stream-static join:

Python

```
@dlt.table

def customer_sales():

  return dlt.read_stream("sales").join(dlt.read("customers"), ["customer_id"], "left")
```

SQL

```
CREATE OR REFRESH STREAMING TABLE customer_sales

AS SELECT * FROM STREAM(LIVE.sales)

  INNER JOIN LEFT LIVE.customers USING (customer_id)
```

Calculate aggregates efficiently

You can use streaming tables to incrementally calculate simple distributive aggregates like count, min, max, or sum, and algebraic aggregates like average or standard deviation. Databricks recommends incremental aggregation for queries with a limited number of groups, for example, a query with a GROUP BY country clause. Only new input data is read with each update.