Last updated on ***Nov 11, 2025***

Tutorial: Build an ETL pipeline with Lakeflow Spark Declarative Pipelines

This tutorial explains how to create and deploy an ETL (extract, transform, and load) pipeline for data orchestration using Lakeflow Spark Declarative Pipelines and Auto Loader. An ETL pipeline implements the steps to read data from source systems, transform that data based on requirements, such as data quality checks and record de-duplication, and write the data to a target system, such as a data warehouse or a data lake.

In this tutorial, you will use pipelines and Auto Loader to:

- Ingest raw source data into a target table.
- Transform the raw source data and write the transformed data to two target materialized views.
- Query the transformed data.
- Automate the ETL pipeline with a Databricks job.

For more information about pipelines and Auto Loader, see Lakeflow Spark Declarative Pipelines and What is Auto Loader?

Requirements

To complete this tutorial, you must meet the following requirements:

- Be logged into a Databricks workspace.
- Have Unity Catalog enabled for your workspace.
- Have serverless compute enabled for your account. Serverless Lakeflow Spark Declarative Pipelines are not available in all workspace regions. See Features with limited regional

availability for available regions.

- Have permission to create a compute resource or access to a compute resource.
- Have permissions to create a new schema in a catalog. The required permissions are `ALL PRIVILEGES` or `USE CATALOG` and `CREATE SCHEMA`.
- Have permissions to create a new volume in an existing schema. The required permissions are `ALL PRIVILEGES` or `USE SCHEMA` and `CREATE VOLUME`.

About the dataset

The dataset used in this example is a subset of the Million Song Dataset, a collection of features and metadata for contemporary music tracks. This dataset is available in the sample datasets included in your Databricks workspace.

Step 1: Create a pipeline

First, create an pipeline by defining the datasets in files (called *source code*) using pipeline syntax. Each source code file can contain only one language, but you can add multiple language-specific files in the pipeline. To learn more, see Lakeflow Spark Declarative Pipelines

This tutorial uses serverless compute and Unity Catalog. For all configuration options that are not specified, use the default settings. If serverless compute is not enabled or supported in your workspace, you can complete the tutorial as written using default compute settings.

To create a new pipeline, follow these steps:

1. In your workspace, click + **New** in the sidebar, then select **ETL Pipeline**.
2. Give your pipeline a unique name.
3. Just below the name, select the default catalog and schema for the data that you generate. You can specify other destinations in your transformations, but this tutorial uses these defaults. You must have permissions to the catalog and schema that you create. See Requirements.
4. For this tutorial, select **Start with an empty file**.
5. In **Folder path**, specify a location for your source files, or accept the default (your user folder).

6. Choose **Python** or **SQL** as the language for your first source file (a pipeline can mix and match languages, but each file must be in a single language).
7. Click **Select**.

The pipeline editor appears for the new pipeline. An empty source file for your language is created, ready for your first transformation.

Step 2: Develop your pipeline logic

In this step, you will use the Lakeflow Pipelines Editor to develop and validate source code for the pipeline interactively.

The code uses Auto Loader for incremental data ingestion. Auto Loader automatically detects and processes new files as they arrive in cloud object storage. To learn more, see What is Auto Loader?

A blank source code file is automatically created and configured for the pipeline. The file is created in the transformations folder of your pipeline. By default, all *.py and *.sql files in the transformations folder are part of the source for your pipeline.

1. Copy and paste the following code into your source file. Be sure to use the language that you selected for the file in Step 1.

Python SQL

Python

```
# Import modules
from pyspark import pipelines as dp
from pyspark.sql.functions import *
from pyspark.sql.types import DoubleType, IntegerType, StringType, StructType,
StructField

# Define the path to the source data
file_path = f"/databricks-datasets/songs/data-001/"

# Define a streaming table to ingest data from a volume
schema = StructType(
  [
```

Ask Assistant

```
    StructField("artist_id", StringType(), True),
    StructField("artist_lat", DoubleType(), True),
    StructField("artist_long", DoubleType(), True),
    StructField("artist_location", StringType(), True),
    StructField("artist_name", StringType(), True),
    StructField("duration", DoubleType(), True),
    StructField("end_of_fade_in", DoubleType(), True),
    StructField("key", IntegerType(), True),
    StructField("key_confidence", DoubleType(), True),
    StructField("loudness", DoubleType(), True),
    StructField("release", StringType(), True),
    StructField("song_hotnes", DoubleType(), True),
    StructField("song_id", StringType(), True),
    StructField("start_of_fade_out", DoubleType(), True),
    StructField("tempo", DoubleType(), True),
    StructField("time_signature", DoubleType(), True),
    StructField("time_signature_confidence", DoubleType(), True),
    StructField("title", StringType(), True),
    StructField("year", IntegerType(), True),
    StructField("partial_sequence", IntegerType(), True)
  ]
)

@dp.table(
  comment="Raw data from a subset of the Million Song Dataset; a collection of
features and metadata for contemporary music tracks."
)
def songs_raw():
  return (spark.readStream
    .format("cloudFiles")
    .schema(schema)
    .option("cloudFiles.format", "csv")
    .option("sep","\t")
    .load(file_path))

# Define a materialized view that validates data and renames a column
@dp.materialized_view(
  comment="Million Song Dataset with data cleaned and prepared for analysis."
)
@dp.expect("valid_artist_name", "artist_name IS NOT NULL")
@dp.expect("valid_title", "song_title IS NOT NULL")
@dp.expect("valid_duration", "duration > 0")
def songs_prepared():
  return (
    spark.read.table("songs_raw")
      .withColumnRenamed("title", "song_title")
      .select("artist_id", "artist_name", "duration", "release", "tempo",
"time_signature", "song_title", "year")
  )
```

```
# Define a materialized view that has a filtered, aggregated, and sorted view of the
data
@dp.materialized_view(
  comment="A table summarizing counts of songs released by the artists who released
the most songs each year."
)
def top_artists_by_year():
  return (
    spark.read.table("songs_prepared")
      .filter(expr("year > 0"))
      .groupBy("artist_name", "year")
      .count().withColumnRenamed("count", "total_number_of_songs")
      .sort(desc("total_number_of_songs"), desc("year"))
  )
```

This source includes code for three queries. You could also put those queries in separate files, to organize the files and code the way that you prefer.

2. Click ▶ **Run file** or **Run pipeline** to start an update for the connected pipeline. With only one source file in your pipeline, these are functionally equivalent.

When the update completes, the editor is updated with information about your pipeline.

- The pipeline graph (DAG), in the sidebar to the right of your code, shows three tables, `songs_raw`, `songs_prepared`, and `top_artists_by_year`.
- A summary of the update is shown at the top of the pipeline assets browser.
- Details of the tables that were generated are shown in the bottom pane, and you can browse data from the tables by selecting one.

This includes the raw and cleaned up data, as well as some simple analysis to find the top artists by year. In the next step, you create ad-hoc queries for further analysis in a separate file in your pipeline.

Step 3: Explore the datasets created by your pipeline

In this step, you perform ad-hoc queries on the data processed in the ETL pipeline to analyze the song data in the Databricks SQL Editor. These queries use the prepared records created in

the previous step.

First, run a query that finds the artists who have released the most songs each year since 1990.

1. From the pipeline assets browser sidebar, click + **Add** then **Exploration**.

2. Enter a **Name** and select **SQL** for the exploration file. A SQL notebook is created in a new `explorations` folder. Files in the `explorations` folder are not run as part of a pipeline update by default. The SQL notebook has cells that you can run together or separately.

3. To create a table of artists that release the most songs in each year after 1990, enter the following code in the new SQL file (if there is sample code in the file, replace it). Because this notebook is not part of the pipeline, it does not use the default catalog and schema. Replace the `<catalog>.<schema>` with the catalog and schema that you used as defaults for the pipeline:

   SQL

   ```
   -- Which artists released the most songs each year in 1990 or later?
   SELECT artist_name, total_number_of_songs, year
     -- replace with the catalog/schema you are using:
     FROM <catalog>.<schema>.top_artists_by_year
     WHERE year >= 1990
     ORDER BY total_number_of_songs DESC, year DESC;
   ```

4. Click ▶ or press `Shift + Enter` to run this query.

Now, run another query that finds songs with a 4/4 beat and danceable tempo.

1. Add the following code to the next cell in the same file. Again, replace the `<catalog>.<schema>` with the catalog and schema that you used as defaults for the pipeline:

   SQL

   ```
   -- Find songs with a 4/4 beat and danceable tempo
   SELECT artist_name, song_title, tempo
     -- replace with the catalog/schema you are using:
     FROM <catalog>.<schema>.songs_prepared
     WHERE time_signature = 4 AND tempo between 100 and 140;
   ```

   Ask Assistant

2. Click ▶ or press `Shift + Enter` to run this query.

Step 4: Create a job to run the pipeline

Next, create a workflow to automate data ingestion, processing, and analysis steps using a Databricks job that runs on a schedule.

1. At the top of the editor, choose the **Schedule** button.
2. If the **Schedules** dialog appears, choose **Add schedule**.
3. This opens the **New schedule** dialog, where you can create a job to run your pipeline on a schedule.
4. Optionally, give the job a name.
5. By default, the schedule is set to run once per day. You can accept this defaut, or set your own schedule. Choosing **Advanced** gives you the option to set a specific time that the job will run. Selecting **More options** allows you to create notifications when the job runs.
6. Select **Create** to apply the changes and create the job.

Now the job will run daily to keep your pipeline up to date. You can choose **Schedule** again to view the list of schedules. You can manage schedules for your pipeline from that dialog, including adding, editing, or removing schedules.

Clicking the name of the schedule (or job) takes you to the job's page in the **Jobs & pipelines** list. From there you can view details about job runs, including the history of runs, or run the job immediately with the **Run now** button.

See Monitoring and observability for Lakeflow Jobs for more information about job runs.

Learn more

- To learn more about data processing pipelines, see Lakeflow Spark Declarative Pipelines
- To learn more about Databricks Notebooks, see Databricks notebooks.
- To learn more about Lakeflow Jobs, see What are jobs?
- To learn more about Delta Lake, see What is Delta Lake in Databricks?