The code below presents a sample DLT notebook containing three sections of scripts for the three stages in the ELT process for this pipeline. The first section will create a live table on your raw data. The format of the source data can be delta, parquet, csv, json and more. The source data can be linked to streaming data flowing into your Delta Lake from cloud\_files and Auto loader sources. Once the first level of the DLT script runs, it will run the next dependent level of the pipeline which creates a live table for your staged data. In this scenario, the script defines expectations that the VendorID is not null and that the passenger\_count is greater than 0 using the EXPECT command. If a violation occurs that does not meet these criteria, those rows will fail to be inserted into this table as a result of the ON VIOLATION command. FAIL UPDATE will immediately stop pipeline execution, whereas DROP ROW will drop the record and continue processing. The EXPECT function can be used at any stage of the pipeline. The select statements in this staging section can be further customized to include joins, aggregations, data cleansing and more. The final level of the DLT script will curate and prepare the final Fact table and will be dependent on the previous staging table script. This wholistic script defines the end-to-end ELT multi staged flow from taking raw data to updating a final consumption layer fact table.

CREATE LIVE TABLE nyctaxi\_raw

COMMENT "This is the raw nyctaxi dataset in Delta Format."

SELECT \* FROM delta. `/mnt/raw/delta/Factnyctaxi`

CREATE LIVE TABLE Factnyctaxi\_staging(

CONSTRAINT valid\_VendorID EXPECT (VendorID IS NOT NULL),

CONSTRAINT valid\_passenger\_count EXPECT (passenger\_count > 0) ON VIOLATION DROP ROW

)

COMMENT "nyctaxi data cleaned and prepared for analysis."

AS SELECT

VendorID AS ID,

CAST(passenger\_count AS INT) AS Count,

total\_amount AS Amount,

trip\_distance AS Distance,

tpep\_pickup\_datetime AS PickUp\_Datetime,

tpep\_dropoff\_datetime AS DropOff\_Datetime

FROM live.nyctaxi\_raw

CREATE LIVE TABLE Factnyctaxi

COMMENT "The curated Factnyc table containing aggregated counts, amounts, and distance data."

AS SELECT

VendorID AS ID,

tpep\_pickup\_datetime AS PickUp\_Datetime,

tpep\_dropoff\_datetime AS DropOff\_Datetime,

CAST(passenger\_count AS INT) AS Count,

total\_amount AS Amount,

trip\_distance AS Distance

FROM live.Factnyctaxi\_staging

WHERE tpep\_pickup\_datetime BETWEEN '2019-03-01 00:00:00' AND '2020-03-01 00:00:00'

AND passenger\_count IS NOT NULL

GROUP BY VendorID, tpep\_pickup\_datetime, tpep\_dropoff\_datetime, CAST(passenger\_count AS INT), total\_amount, trip\_distance

ORDER BY VendorID ASC

This SQL code could just as easily be written in Python if needed. You’ll first need to run commands similar to the following script shown below to import delta live tables along with PySpark SQL functions and types.

import dlt

from pyspark.sql.functions import \*

from pyspark.sql.types import \*

Similar to the SQL EXPECT function in the SQL DLT pipeline notebook script above, the following commands can be used within PySpark to handle row violations based on the expectations:

* **expect\_or\_drop:** If a row violates the expectation, drop the row from the target dataset. For example, the following code can be used to expand upon the original expectation that we created to now drop rows with VendorIDs that are null by adding the ON VIOLATION DROP ROW command: CONSTRAINT valid\_VendorID EXPECT (VendorID IS NOT NULL)ON VIOLATION DROP ROW
* **expect\_or\_fail:** If a row violates the expectation, immediately stop execution. For example, the following code can be used to expand upon the original expectation that we created to now fail the update if there are null VendorIDs by adding the ON VIOLATION FAIL UPDATE command: CONSTRAINT valid\_VendorID EXPECT (VendorID IS NOT NULL)ON VIOLATION FAIL UPDATE
* **expect\_all:** If a row violates any of the expectations, include the row in the target dataset. For example, the following line of Python code contains the various expect\_all conditions: @dlt.expect\_all({"valid\_VendorID": "VendorID IS NOT NULL”, "valid\_VendorName": "VendorName IS NOT NULL"}). The code shown below includes this in a sample by defining valid\_Vendor as a data frame and calling it it with the expect\_all violation commands.
* **expect\_all\_or\_drop:** If a row violates any of the expectations, drop the row from the target dataset. This can be seen in the code shown below within the section showing the following code line: @dlt.expect\_all\_or\_drop(valid\_Vendor)
* **expect\_all\_or\_fail:** If a row violates any of the expectations, immediately stop execution. Similar to the expect\_all\_or\_drop code line as shown in the code below, this expect\_all\_or\_fail command can be interchanged or added when needed.

valid\_Vendor = {"valid\_VendorID": "VendorID IS NOT NULL”, "valid\_VendorName": "VendorName IS NOT NULL"}

@dlt.table

@dlt.expect\_all(valid\_Vendor)

def raw\_data():

# Create raw dataset

@dlt.table

@dlt.expect\_all\_or\_drop(valid\_Vendor)

def curated\_data():

# Create cleaned and prepared dataset

As an example, here is what the pipeline’s JSON script would look like. This JSON can be further customized as needed.

{

"name": "DLT NYCTaxi Data Pipeline",

"storage": "/mnt/data/raw/Factnyctaxi",

"clusters": [

{

"num\_workers": 1,

"spark\_conf": {}

}

],

"libraries": [

{

"notebook": {

"path": "/Users/ronlesteve/dlt/Factnyctaxi

}

}

],

"continuous": false

}

From a use case perspective, for example, if a developer has two different ETL processes that require different parameters but need to be tied to the same pipeline, they could run multiple notebooks within a pipeline, each containing their own parameters. The following code shows the JSON code to connect multiple notebooks within a pipeline.

{

"name": "DLT\_Pipeline",

"storage": "dbfs:/data/raw/vendors",

"libraries": [

{ "notebook": { "path": "/etl\_1" } },

{ "notebook": { "path": "/etl\_2" } }

]

}

This next block of SQL code shows how to parameterize pipelines within the notebooks.

CREATE LIVE TABLE vendors

AS SELECT \* FROM srcVendors WHERE date > ${DLT\_pipeline.startDate};

As an example, the code below creates a view for the system event metrics for the data that has been processed using the DLT pipeline.

ADLSg2Path = “/mnt/raw/data/NycTaxidata”

df = spark.read.format("delta").load(f"{ADLSg2Path}/system/events")

df.createOrReplaceTempView("dlteventmetrics")

Once the view is created, you can simply write PySpark or SQL scripts similar to the code shown below to display the metrics related to audit logs.

SELECT

timestamp,

details:user\_action:action,

details:user\_action:user\_name

FROM event\_log\_raw

WHERE event\_type = 'user\_action'

This next query is more complex and can be created on the same view to explode the nested JSON array contents to extract a more customized report on the quality of the data based on the expectations for passing and failing of the rows.

SELECT

row\_expectations.dataset as dataset,

row\_expectations.name as expectation,

SUM(row\_expectations.passed\_records) as passing\_records,

SUM(row\_expectations.failed\_records) as failing\_records

FROM

(

SELECT

explode(

from\_json(

details :flow\_progress :data\_quality :expectations,

"array<struct<name: string, dataset: string, passed\_records: int, failed\_records: int>>"

)

) row\_expectations

FROM

dlteventmetrics

WHERE

event\_type = 'flow\_progress'

AND origin.update\_id = '${latest\_update.id}'

)

GROUP BY

row\_expectations.dataset,

row\_expectations.name