≫ Delta Live Tables uses Delta Lake to store all tables. Each time a query is executed to fetch results from a DLT, the most recent version of the table is returned.

≫ When we apply an APPLY INTO operation on a DLT, DLT uses a hidden table with a name as __apply_changes_storage_<DLT_table_name to which changes are applied>

It is used by DLT to ensure updates are applied in the correct order to materialize results correctly. Our target SCD1 table is actually shown using a VIEW registered on this hidden table.

≫ The Event logs of the events of changes occuring to a DLT are stored as Delta Tables.

Delta Live Table folder

Created when a DLT is created using Workflow Pipeline

sub-directories

- /autoloder/
- /checkpoints/
- /system/
- /tables/

Both these directories contain data used to manage incremental data processing with Structured Streaming

/system/ : Captures events associated with the pipeline (in the form of Delta Tables)

/tables/ : Contains Delta Tables managed by DLT

Event Logs :-

contains all data in tabular format, that is shown in the Pipeline UI page.

DLT uses event logs to store most of the important information used to manage, report and understand what is happening during pipeline execution.

If we have created a Delta Live Table in a location contained in, say location parameter then we can query the event log of the DLT as follow.

```
SELECT * FROM delta.`${location}/system/events"
```

inside this folder structure exists the event logs of the DLT.

The event logs is managed as a Delta Lake Table, with some important fields stored as nested JSON data.

Set Latest Update ID

In many cases, you may wish to gain updates about the latest update (or the last N updates) to your pipeline.

We can easily capture the most recent update ID with a SQL query.

md 7

```
latest_update_id = spark.sql("""
    SELECT origin.update_id
    FROM event_log_raw
    WHERE event_type = 'create_update'
    ORDER BY timestamp DESC LIMIT 1""").first().update_id
print(f"Latest Update ID: {latest_update_id}")

# Push back into the spark config so that we can use it in a later query.
spark.conf.set('latest_update.id', latest_update_id)
```

This table is being stored as a view by this name :- event_log_raw

accessing child item "update_id" of the struct object "origin"

⇒ For syntaxes, and below topics, refer to the DataEngg. v3 notebook: DE 4.1.1 - Orders Pipeline

Imp:-

⇒ DLT codes and syntaxes are not for interactive execution in a notebook, rather, to a part of execution through DLT pipeline.

⇒ In DLT code, we use parameters, value values to those parameters are passed at the Pipeline level, when the pipeline is triggered to run (similar to parameters passing in Az. Data Factory)

⇒ Types of Persistent tables created with DLT :-

i) Live Tables : materialized view for lakehouse, return current results of any query, with each refresh

ii) Streaming Live Tables : for incremental, near-real time data processing.

⇒ Basic Syntax for SQL DLT :-

```
CREATE OR REFRESH [STREAMING] LIVE TABLE <tbl_name>
AS <select statement>
```

⇒ Both types of DLT are persisted as tables in Delta Lake protocol.

Streaming Ingestion with Auto Loader

⇒ Auto Loader helps to incrementally load data from cloud object storage into Delta Lake.

⇒ To use Auto Loader in DLT
- ⇒ Configure a source directory,
- ⇒ Provide a few configuration settings,
- ⇒ Write/define a query against source.

⇒ Auto Loader automatically detects new files being added to source, and incrementally processed only the new records.

⇒ The `cloud_files()` function enables Auto Loader to be used natively in SQL. It takes follow params:
1. Source location — can be cloud object storage,
2. Data format in source (Ex→ json, parquet,..)
3. Arbitary-sized comma-seperated list of optional reader options: Ex:- `map("cloudFiles.inferColumnTypes", "true")`
   - used to infer the schema, datatypes of columns automatically

```
CREATE OR REFRESH STREAMING LIVE TABLE orders_bronze
AS SELECT current_timestamp() processing_time, input_file_name() source_file, *
FROM cloud_files("${source}/orders", "json", map("cloudFiles.inferColumnTypes", "true"))
```

should be a path to a folder (not any particular file)

⇒ Cloud Files actually returns tabular data, that can be actually used in place of table in SELECT statement

Ex:-

ON VIOLATE FAIL ROW

OR

```
CREATE OR REFRESH STREAMING LIVE TABLE orders_silver
(CONSTRAINT valid_date EXPECT (order_timestamp > "2021-01-01") ON VIOLATION FAIL UPDATE)
COMMENT "Append only orders with valid timestamps"
TBLPROPERTIES ("quality" = "silver")
AS SELECT timestamp(order_timestamp) AS order_timestamp, * EXCEPT (order_timestamp, source_file, _rescued_data)
FROM STREAM(LIVE.orders_bronze)
```

LIVE refers to that database that is set as target database, in the DLT pipeline (Using DLT keyword is must for referencing a DLT)

STREAM makes the table (in the parameter) a streaming source of data

Live Tables vs. Streaming Live Tables

The two queries we've reviewed so far have both created streaming live tables. Below, we see a simple query that returns a live table (or materialized view) of some aggregated data.

Spark has historically differentiated between batch queries and streaming queries. Live tables and streaming live tables have similar differences.

Note the only syntactic differences between streaming live tables and live tables are the lack of the `STREAMING` keyword in the create clause and not wrapping the source table in the `STREAM()` method.

Below are some of the differences between these types of tables.

Live Tables

- Always "correct", meaning their contents will match their definition after any update.
- Return same results as if table had just been defined for first time on all data.
- Should not be modified by operations external to the DLT Pipeline (you'll either get undefined answers or your change will just be undone).

Streaming Live Tables

- Only supports reading from "append-only" streaming sources.
- Only reads each input batch once, no matter what (even if joined dimensions change, or if the query definition changes, etc).
- Can perform operations on the table outside the managed DLT Pipeline (append data, perform GDPR, etc).

```
CREATE OR REFRESH LIVE TABLE orders_by_date
AS SELECT date(order_timestamp) AS order_date, count(*) AS total_daily_orders
FROM LIVE.orders_silver
GROUP BY date(order_timestamp)
```

For aggregations, general LIVE tables are used, as each time, they process the entire source data.

Processing CDC Data with `APPLY CHANGES INTO`

DLT introduces a new syntactic structure for simplifying CDC feed processing.

`APPLY CHANGES INTO` has the following guarantees and requirements:

- Performs incremental/streaming ingestion of CDC data
- Provides simple syntax to specify one or many fields as the primary key for a table
- Default assumption is that rows will contain inserts and updates
- Can optionally apply deletes
- Automatically orders late-arriving records using user-provided sequencing key
- Uses a simple syntax for specifying columns to ignore with the `EXCEPT` keyword
- Will default to applying changes as Type 1 SCD

The code below:

- Creates the `customers_silver` table; `APPLY CHANGES INTO` requires the target table to be declared in a separate statement
- Identifies the `customers_silver` table as the target into which the changes will be applied
- Specifies the table `customers_bronze_clean` as the streaming source
- Identifies the `customer_id` as the primary key
- Specifies that records where the `operation` field is `DELETE` should be applied as deletes
- Specifies the `timestamp` field for ordering how operations should be applied
- Indicates that all fields should be added to the target table except `operation`, `source_file`, and `_rescued_data`

```
CREATE OR REFRESH STREAMING LIVE TABLE customers_silver;

APPLY CHANGES INTO LIVE.customers_silver
  FROM STREAM(LIVE.customers_bronze_clean)
  KEYS (customer_id)
  APPLY AS DELETE WHEN operation = "DELETE"
  SEQUENCE BY timestamp
  COLUMNS * EXCEPT (operation, source_file, _rescued_data)
```

DLT Views :-

=> CREATE LIVE VIEW <view_name>
. . . . (rest same)

=> Views in DLT are not persisted in Databricks metastore, and can only be accessed & used from within the DLT pipeline.
Thus, DLT views can be thought of as Temporary general views.

=> DLT views are not persistent (not stored on disk). They are used as temporary tables, accessible only inside the DLT pipeline which creates it.

Adding multiple conditions to a single constraint, and multiple constraints to a single query:-

```
CREATE STREAMING LIVE TABLE customers_bronze_clean
(CONSTRAINT valid_id EXPECT (customer_id IS NOT NULL) ON VIOLATION FAIL UPDATE,
CONSTRAINT valid_operation EXPECT (operation IS NOT NULL) ON VIOLATION DROP ROW,
CONSTRAINT valid_name EXPECT (name IS NOT NULL or operation = "DELETE"),
CONSTRAINT valid_address EXPECT (
  (address IS NOT NULL and
  city IS NOT NULL and
  state IS NOT NULL and
  zip_code IS NOT NULL) or
  operation = "DELETE"),
CONSTRAINT valid_email EXPECT (
  rlike(email, '^([a-zA-Z0-9_\\-\\.]+)@([a-zA-Z0-9_\\-\\.]+)\\.([a-zA-Z]{2,5})$') or
  operation = "DELETE") ON VIOLATION DROP ROW)
AS SELECT *
  FROM STREAM(LIVE.customers_bronze)
```

When no ON VIOLATION is mentioned, that means the records not matching the condition will only be tracked/logged, but its insertion will occur into target successfully.

Applying CDC → SCD Type 1 Load

Processing CDC Data with `APPLY CHANGES INTO`

DLT introduces a new syntactic structure for simplifying CDC feed processing.

`APPLY CHANGES INTO` has the following guarantees and requirements:

- Performs incremental/streaming ingestion of CDC data
- Provides simple syntax to specify one or many fields as the primary key for a table
- Default assumption is that rows will contain inserts and updates
- Can optionally apply deletes
- Automatically orders late-arriving records using user-provided sequencing key
- Uses a simple syntax for specifying columns to ignore with the `EXCEPT` keyword
- Will default to applying changes as Type 1 SCD

The code below:

- Creates the `customers_silver` table; `APPLY CHANGES INTO` requires the target table to be declared in a separate statement
- Identifies the `customers_silver` table as the target into which the changes will be applied
- Specifies the table `customers_bronze_clean` as the streaming source
- Identifies the `customer_id` as the primary key
- Specifies that records where the `operation` field is `DELETE` should be applied as deletes
- Specifies the `timestamp` field for ordering how operations should be applied
- Indicates that all fields should be added to the target table except `operation`, `source_file`, and `_rescued_data`

```
CREATE OR REFRESH STREAMING LIVE TABLE customers_silver;

APPLY CHANGES INTO LIVE.customers_silver
  FROM STREAM(LIVE.customers_bronze_clean)
  KEYS (customer_id)
  APPLY AS DELETE WHEN operation = "DELETE"
  SEQUENCE BY timestamp
  COLUMNS * EXCEPT (operation, source_file, _rescued_data)
```

The column(s) specified in the KEYS clause is the primary key column. There will be at most 1 record for each unique value of this column.

For records with same value in the primary key column, record with the latest/greatest value in the column

specified in the SEQUENCE clause will be kept in the target table.

⇒ Application of an APPLY CHANGES INTO TABLE :-

Querying Tables with Applied Changes

`APPLY CHANGES INTO` defaults to creating a Type 1 SCD table, meaning that each unique key will have at most 1 record and that updates will overwrite the original information.

While the target of our operation in the previous cell was defined as a streaming live table, data is being updated and deleted in this table (and so breaks the append-only requirements for streaming live table sources). As such, downstream operations cannot perform streaming queries against this table.

This pattern ensures that if any updates arrive out of order, downstream results can be properly recomputed to reflect updates. It also ensures that when records are deleted from a source table, these values are no longer reflected in tables later in the pipeline.

Below, we define a simple aggregate query to create a live table from the data in the `customers_silver` table.

```
CREATE LIVE TABLE customer_counts_state
  COMMENT "Total active customers per state"
AS SELECT state, count(*) as customer_count, current_timestamp() updated_at
  FROM LIVE.customers_silver
  GROUP BY state
```

⇒ Apply changes into loading into a Live Table is useful, when status of entities in the table changes, and we want to perform aggregation based on the changing status.

⇒ Target LIVE TABLE of APPLY CHANGES INTO cannot act as Streaming source for any streaming

live table, because the table becomes and SCD Type 1 table (and not an append-only table).

Thus, APPLY CHANGES INTO table cannot be used inside the STREAM() function.