DLT Syntaxes in Python :-

⇒ To use any DLT syntaxes in Python, such as creating or manipulating Delta Live Tables, we need to import dlt module.

```
import dlt
```

Tables as DataFrames

There are two distinct types of persistent tables that can be created with DLT:

- **Live tables** are materialized views for the lakehouse; they will return the current results of any query with each refresh
- **Streaming live tables** are designed for incremental, near-real time data processing

Note that both of these objects are persisted as tables stored with the Delta Lake protocol (providing ACID transactions, versioning, and many other benefits). We'll talk more about the differences between live tables and streaming live tables later in the notebook.

Delta Live Tables introduces a number of new Python functions that extend familiar PySpark APIs.

At the heart of this design, the decorator `@dlt.table` is added to any Python function that returns a Spark DataFrame. (**NOTE**: This includes Koalas DataFrames, but these won't be covered in this course.)

If you're used to working with Spark and/or Structured Streaming, you'll recognize the majority of the syntax used in DLT. The big difference is that you'll never see any methods or options for DataFrame writers, as this logic is handled by DLT.

As such, the basic form of a DLT table definition will look like:

```
@dlt.table
def <function-name>():
return (<query>)
```

⇒ Inside a Delta Live Table creation function, we only can write the logic for table creation and data population.

No Dataframe writing is written, as the

writing is automatically handled by DLT pipeline. All the tables are written inside the Target Schema set during pipeline creation.

Thus, to control the location of storage of DL Tables, we just make the schema in the desired location.

Table Name of the Delta Live Table will be name of the Python function that creates it.

Streaming Ingestion with Auto Loader

Databricks has developed the Auto Loader functionality to provide optimized execution for incrementally loading data from cloud object storage into Delta Lake. Using Auto Loader with DLT is simple: just configure a source data directory, provide a few configuration settings, and write a query against your source data. Auto Loader will automatically detect new data files as they land in the source cloud object storage location, incrementally processing new records without the need to perform expensive scans and recomputing results for infinitely growing datasets.

Auto Loader can be combined with Structured Streaming APIs to perform incremental data ingestion throughout Databricks by configuring the `format("cloudFiles")` setting. In DLT, you'll only configure settings associated with reading data, noting that the locations for schema inference and evolution will also be configured automatically if those settings are enabled.

The query below returns a streaming DataFrame from a source configured with Auto Loader.

In addition to passing `cloudFiles` as the format, here we specify:

- The option `cloudFiles.format` as `json` (this indicates the format of the files in the cloud object storage location)
- The option `cloudFiles.inferColumnTypes` as `True` (to auto-detect the types of each column)
- The path of the cloud object storage to the `load` method
- A select statement that includes a couple of `pyspark.sql.functions` to enrich the data alongside all the source fields

By default, `@dlt.table` will use the name of the function as the name for the target table.

```
@dlt.table
def orders_bronze():
    return (
        spark.readStream
            .format("cloudFiles")
            .option("cloudFiles.format", "json")
            .option("cloudFiles.inferColumnTypes", True)
            .load(f"{source}/orders")
            .select(
                F.current_timestamp().alias("processing_time"),
                F.input_file_name().alias("source_file"),
                "*"
            )
    )
```

this will be the table name of the STREAMING DLT.

DF returned by this query will be written into the DLT (table will be created first, if non-existent)

Validating & Enriching Data :-

Options for `@dlt.table()`

There are a number of options that can be specified during table creation. Here, we use two of these to annotate our dataset.

`comment`

Table comments are a standard for relational databases. They can be used to provide useful information to users throughout your organization. In this example, we write a short human-readable description of the table that describes how data is being ingested and enforced (which could also be gleaned from reviewing other table metadata).

`table_properties`

This field can be used to pass any number of key/value pairs for custom tagging of data. Here, we set the value `silver` for the key `quality`.

Note that while this field allows for custom tags to be arbitrarily set, it is also used for configuring number of settings that control how a table will perform. While reviewing table details, you may also encounter a number of settings that are turned on by default any time a table is created.

Data Quality Constraints

The Python version of DLT uses decorator functions to set data quality constraints. We'll see a number of these throughout the course.

DLT uses simple boolean statements to allow quality enforcement checks on data. In the statement below, we:

- Declare a constraint named `valid_date`
- Define the conditional check that the field `order_timestamp` must contain a value greater than January 1, 2021
- Instruct DLT to fail the current transaction if any records violate the constraint by using the decorator `@dlt.expect_or_fail()`

Each constraint can have multiple conditions, and multiple constraints can be set for a single table. In addition to failing the update, constraint violation can also automatically drop records or just record the number of violations while still processing these invalid records.

DLT Read Methods

The Python `dlt` module provides the `read()` and `read_stream()` methods to easily configure references to other tables and views in your DLT Pipeline. This syntax allows you to reference these datasets by name without any database reference. You can also use `spark.table("LIVE.<table_name.")`, where `LIVE` is a keyword substituted for the database being referenced in the DLT Pipeline.

```
@dlt.table(
    comment = "Append only orders with valid timestamps",
    table_properties = {"quality": "silver"})
@dlt.expect_or_fail("valid_date", F.col("order_timestamp") > "2021-01-01")
def orders_silver():
    return (
        dlt.read_stream("orders_bronze")
            .select(
                "processing_time",
                "customer_id",
                "notifications",
                "order_id",
                F.col("order_timestamp").cast("timestamp").alias("order_timestamp")
            )
    )
```

syntax to read a streaming DLT.

can also write this line as:

```
spark.table("LIVE.orders_bronze")
    .select
```

==

General DLT (not Streaming)

```python
@dlt.table
def orders_by_date():
    return (
        dlt.read("orders_silver")
            .groupBy(F.col("order_timestamp").cast("date").alias("order_date"))
            .agg(F.count("*").alias("total_daily_orders"))
    )
```

=> Setting LIVE TABLE NAME diff. from the function name :-

Specifying Table Names

The code below demonstrates the use of the `name` option for DLT table declaration. The option allows developers to specify the name for the resultant table separate from the function definition that creates the DataFrame the table is defined from.

In the example below, we use this option to fulfill a table naming convention of `<dataset-name>_<data-quality>` and a function naming convention that describes what the function is doing. (If we hadn't specified this option, the table name would have been inferred from the function as `ingest_customers_cdc`.)

```python
@dlt.table(
    name = "customers_bronze",
    comment = "Raw data from customers CDC feed"
)
def ingest_customers_cdc():
    return (
        spark.readStream
        .format("cloudFiles")
        .option("cloudFiles.format", "json")
        .load(f"{source}/customers")
        .select(
            F.current_timestamp().alias("processing_time"),
            F.input_file_name().alias("source_file"),
            "*"
        )
    )
```

Adding multiple constraints / Condition in single query:

PYTHON syntax

```
@dlt.table
@dlt.expect_or_fail("valid_id", "customer_id IS NOT NULL")
@dlt.expect_or_drop("valid_operation", "operation IS NOT NULL")
@dlt.expect("valid_name", "name IS NOT NULL or operation = 'DELETE'")
@dlt.expect("valid_adress", """
    (address IS NOT NULL and
    city IS NOT NULL and
    state IS NOT NULL and
    zip_code IS NOT NULL) or
    operation = "DELETE"
    """)
@dlt.expect_or_drop("valid_email", """
    rlike(email, '^([a-zA-Z0-9_\\\\-\\\\.]+)@([a-zA-Z0-9_\\\\-\\\\.]+)\\\\.([a-zA-Z]{2,5})$') or
    operation = "DELETE"
    """)
def customers_bronze_clean():
    return (
        dlt.read_stream("customers_bronze")
    )
```

only expect means the violating records will be only tracked and logged, but will be successfully loaded into the target table.

- The conditions of the Expectations are written in SQL style, inside quotes.

dlt.apply_changes() - UPSERT:

SCD Type 1 loading :-

Processing CDC Data with `dlt.apply_changes()`

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

`dlt.apply_changes()` 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 field
- Uses a simple syntax for specifying columns to ignore with the `except_column_list`
- Will default to applying changes as Type 1 SCD

The code below:

- Creates the `customers_silver` table; `dlt.apply_changes()` 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 source (**NOTE**: source must be append-only)
- Identifies the `customer_id` as the primary key
- Specifies the `timestamp` field for ordering how operations should be applied
- Specifies that records where the `operation` field is `DELETE` should be applied as deletes
- Indicates that all fields should be added to the target table except `operation`, `source_file`, and `_rescued_data`

```
dlt.create_target_table(
    name = "customers_silver")

dlt.apply_changes(
    target = "customers_silver",
    source = "customers_bronze_clean",
    keys = ["customer_id"],
    sequence_by = F.col("timestamp"),
    apply_as_deletes = F.expr("operation = 'DELETE'"),
    except_column_list = ["operation", "source_file", "_rescued_data"])
```

=> Which is better ?

Python vs SQL

Python	SQL	Notes
Python API	Proprietary SQL API	
No syntax check	Has syntax checks	In Python, if you run a DLT notebook cell on its own it will show in error, whereas in SQL it will check if the command is syntactically valid and tell you. In both cases, individual notebook cells are not supposed to be run for DLT pipelines.
A note on imports	None	The dlt module should be explicitly imported into your Python notebook libraries. In SQL, this is not the case.
Tables as DataFrames	Tables as query results	The Python DataFrame API allows for multiple transformations of a dataset by stringing multiple API calls together. Compared to SQL, those same transformations must be saved in temporary tables as they are transformed.
@dlt.table()	SELECT statement	In SQL, the core logic of your query, containing transformations you make to your data, is contained in the SELECT statement. In Python, data transformations are specified when you configure options for @dlt.table().
@dlt.table(comment = "Python comment",table_properties = {"quality": "silver"})	COMMENT "SQL comment" TBLPROPERTIES ("quality" = "silver")	This is how you add comments and table properties in Python vs. SQL