Python for Delta Live Tables

Exploring Python's role in data management

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Agenda of the Presentation

- Introduction to Python for DLT
- Why Python for DLT?
- Core DLT Python Decorators: @dlt.table
- Core DLT Python Decorators: @dlt.view
- Defining a DLT Table in Python - Example
- Reading from Other DLT Tables/Views in Python
- Implementing Data Quality with Expectations in Python
- Hands-on Lab: Python Pipeline Creation (Recap for Exam)

Introduction to Python for DLT

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Why Python for DLT?

Flexibility & Power

Flexibility & Power

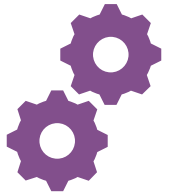
Utilize the full expressiveness of Python for complex transformations.

Access to a rich ecosystem of Python libraries (though direct use within DLT transformations needs care; primarily PySpark API).

Define and use Python UDFs for custom logic not easily achievable with standard Spark functions.

Familiarity for Python Developers

Lower barrier to entry for teams already proficient in Python and PySpark.



Code Organization:

Structure complex pipelines within Python functions and modules.

Familiarity for Python Developers

Python UDFs in Spark

Python UDFs enable the inclusion of custom logic in Spark applications, enhancing functionality beyond standard functions.

Lower Barrier to Entry

Teams proficient in Python and PySpark can easily adopt Spark, reducing learning curves and increasing productivity.

Code Organization

```
@dlt.table(
 name="uc01.sales_new.bronze_sales",
 comment="Raw sales data ingested to Bronze layer",
 table_properties={
  "quality": "bronze"
 }
)
def bronze_sales():
  return (
    spark.read.option("header", "true")
          .option("inferSchema", "true")
          .csv("/FileStore/sales.csv")
  )
```

Structuring Code

Organizing code into functions and modules enhances readability and maintainability.

Complex Pipelines

Using functions helps in managing complex data pipelines in Python effectively.

Exam Relevance

Understanding code organization is essential for successfully completing programming exams and projects.

Core DLT Python Decorators: @dlt.table

Purpose and Syntax

Defining a DLT Table

The purpose of the syntax is to define a DLT table, which materializes its output for further processing.

Syntax Breakdown

The syntax includes several optional parameters such as name, comment, configuration, and properties that enhance table management.

Table Properties

Table properties can determine aspects like quality and custom settings, which can be configured for specific use cases.

```
@dlt.table(
  name="your_table_name",  // Optional: inferred from function name if omitted
  comment="Descriptive comment about the table.",
  table_properties={"quality": "bronze", "custom_key": "custom_value"},
  path="/path/to/custom/storage_location", // Optional: DLT manages by default
  schema="colA STRING, colB INT", // Optional: explicit schema, can be inferred
  temporary=False // False by default
)
def python_function_defining_table():
        return spark.createDataFrame(...) # or spark.read...
```

Core DLT Python Decorators: @dlt.view

Purpose and Syntax

Definition of DLT View

A DLT view is a logical transformation that defines how data is presented, not stored directly.

View Syntax Overview

The syntax allows users to define a view with optional metadata like name and comments for clarity.

```
@dlt.view(
  name="your_view_name", // Optional: inferred
  comment="Descriptive comment about the view."
)
def python_function_defining_view():
  # ... PySpark DataFrame logic ...
  return dlt.read("some_other_table").select(...)
```

Key Differences from @dlt.table

Views vs Tables

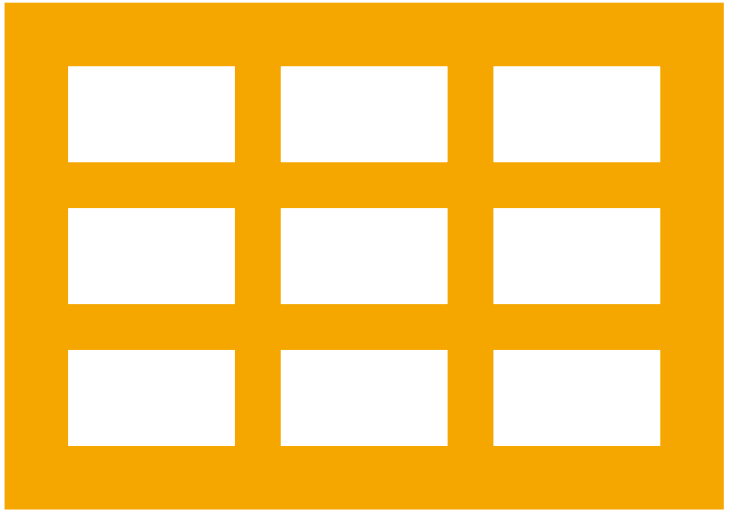
Views do not have a path or schema, unlike tables that are directly stored and materialized.

Usage of Views

Views are ideal for intermediate steps and simplifying complex queries without the overhead of full tables.

Exam Relevance

Understanding the distinction between DLT tables and views is crucial for effective data management and exam success.



Defining a DLT Table in Python - Example

Scenario: Ingesting raw JSON user data into a Bronze table

Introduction to Data Ingestion

This scenario demonstrates how to ingest raw user data from JSON files into a Bronze table for processing.

Using DLT for Data Management

Data Leveraging Technology (DLT) enables seamless integration and management of incoming data streams.

Streaming Data with Spark

Apache Spark facilitates the real-time processing of streaming data, ensuring timely data availability.

```
import dlt
from pyspark.sql.functions import * # Common practice
# Assumes spark is available (provided by DLT environment)
@dlt.table(
  name="bronze_users_python",
  comment="Raw user data ingested from JSON files.",
  table_properties={"quality": "bronze"}
)
def ingest_raw_user_data():
 return (
  spark.readStream.format("cloudFiles")
   .option("cloudFiles.format", "json")
   .option("cloudFiles.schemaLocation",
"/path/to/schema_location_bronze_users") # Important for
cloudFiles
   .load("/mnt/datalake/raw/users/")
 )
```

Key Parameters

```
@dlt.table(
   name="your_table_name",  // Optional: inferred from function name if omitted
   comment="Descriptive comment about the table.",
   spark_conf={"key": "value"},
   table_properties={"quality": "bronze", "custom_key": "custom_value"},
   path="/path/to/custom/storage_location", // Optional: DLT manages by default
   schema="colA STRING, colB INT", // Optional: explicit schema, can be inferred
   temporary=False // False by default
)
def python_function_defining_table():
   # ... PySpark DataFrame logic ...
   return spark.createDataFrame(...) # or spark.read...
```

Table Naming Convention

The 'name' parameter is crucial for referencing this table in other areas of documentation and systems.

Documentation Importance

The 'comment' field aids in documentation and understanding the lineage of the data, making it essential for clarity.

Tagging and Governance

The 'table_properties' are significant for tagging data elements, such as Medallion architecture layers, for governance purposes.

Schema Definition

Understanding how to explicitly define a schema is crucial for ensuring data integrity and correctness.

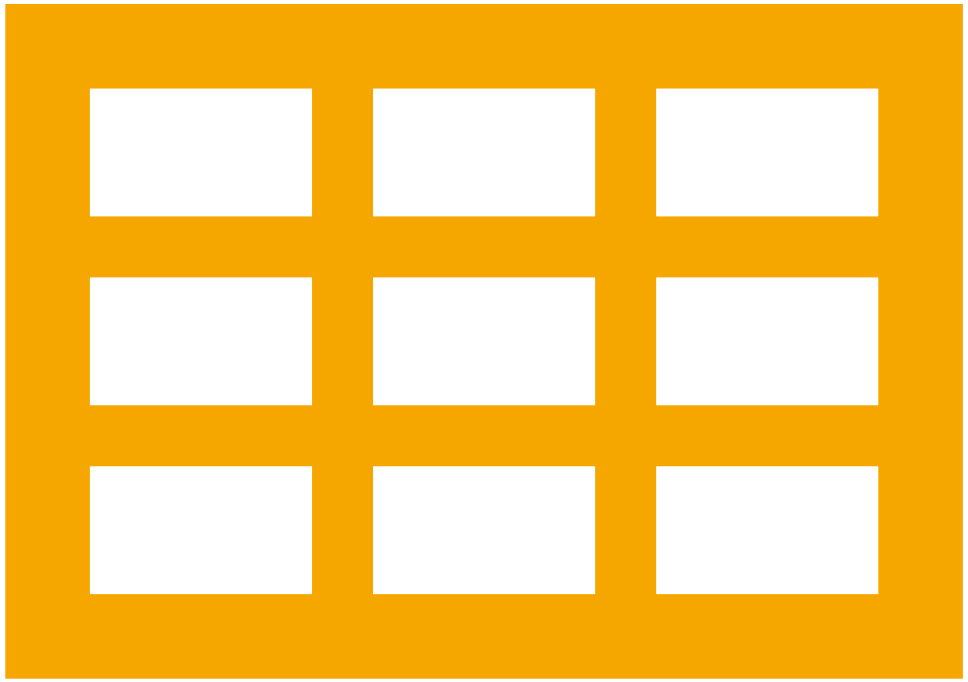
name: Crucial for referencing this table elsewhere.

comment: Important for documentation and understanding lineage.

table_properties: Used for tagging (e.g., Medallion architecture layer), governance.

schema: How to explicitly define a schema if needed.

Reading from Other DLT Tables/Views in Python



```
@dlt.table(
    name="silver_users_python",
    comment="Cleaned and transformed user data.",
    table_properties={"quality": "silver"}
)
def process_silver_user_data():
  bronze_df = dlt.read_stream("bronze_users_python")
  return (
    bronze_df
      .withColumn("registration_date", to_date(col("registration_ts"),
"yyyy-MM-dd"))
      .withColumn("email_domain", split(col("email"), "@")[1])
      .filter(col("email").isNotNull())
  )
```

Mechanism and Example

Data Reading Mechanism

Use dlt.read() for batch tables and dlt.read_stream() for streaming data to efficiently manage inputs.

Example of Silver Table Creation

This example demonstrates how to create a Silver table from a Bronze table, transforming user data effectively.

Key Point

Understanding DLT Dependencies

The command dlt.read_stream establishes a crucial dependency in data processing, ensuring that data flows correctly within the pipeline.

Reading Upstream Datasets

It's essential to know how to read from upstream DLT datasets to effectively manage data flows and dependencies in Python.

```
dlt.read_stream("bronze_users_python")
```

Python DLT Pipeline

Creating and managing dependencies in a Python DLT pipeline is crucial for exam relevance and practical applications in data engineering.

Implementing Data Quality with Expectations in Python

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Purpose and Syntax

Defining Data Quality Rules

Data quality rules are essential for ensuring the accuracy and reliability of data in your pipeline.

Using Decorators

Decorators are used to implement data quality rules, enabling logging, dropping, or failing records based on conditions.

Types of Expectations

Understand different types of expectations–log, drop, or fail–and their respective actions on data validation.

Decorators & Syntax:

@dlt.expect("constraint_name", "boolean_condition_as_string"): Records failing are logged.

Example: @dlt.expect("valid_email_format", "email RLIKE '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\\\.[a-zA-Z]{2,}'")

@dlt.expect_or_drop("constraint_name", "condition"): Records failing are dropped.

Example: @dlt.expect_or_drop("user_id_present", "user_id IS NOT NULL")

@dlt.expect_or_fail("constraint_name", "condition"): Pipeline run fails if records violate.

Example: @dlt.expect_or_fail("critical_status_check", "status = 'ACTIVE'")

@dlt.expect_all({"name1": "cond1", "name2": "cond2"}): Apply multiple expectations, all logged.

@dlt.expect_all_or_drop({...})

@dlt.expect_all_or_fail({...})

Operational Excellence: Running Modes, Results, and Lineage

DLT Pipeline Running Modes

Development Mode

Purpose: Iterative development, testing, debugging.

Key Characteristics:

Cluster Reuse: Can reuse clusters to speed up iterative runs (configurable).

Error Handling: May halt sooner on certain non-fatal errors.

Configuration Flexibility: Easier to make frequent changes.

- **No Retries by Default:** Pipeline does not automatically retry on failure.

Production Mode:

Purpose: Reliable, automated execution of validated pipelines.

Key Characteristics:

New Cluster Always: Ensures a clean, isolated environment for each run.

Enhanced Retries: Automatic retries on transient failures.

- **Optimized for Throughput/Cost:** Cluster settings often geared towards efficiency.

Switching Modes: Done in the DLT pipeline settings UI.

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

DLT Execution Triggers: Continuous vs. Triggered

Triggered Mode:	**Behavior:** Processes all available data since the last run, then stops the cluster (if configured to terminate).
	Use Cases: Batch processing, periodic updates (e.g., nightly, hourly).
	Cost: Can be more cost-effective if data arrives infrequently, as cluster isn't always on.

Continuous Mode:	**Behavior:** Cluster remains active, continuously ingesting and processing new data as it arrives at the source.
	Use Cases: Real-time or near real-time streaming applications.
	Sources: Requires streaming sources (e.g., Auto Loader with cloudFiles, Kafka).
	State Management: DLT handles state for aggregations, joins across micro-batches.
	Cost: Higher compute cost due to the always-on cluster.
	Watermarking: Important for managing late data in stateful streaming operations (though DLT abstracts some of this).

Analyzing Pipeline Results in the DLT UI

Pipeline Graph (DAG):

- Visual representation of tables/views and their dependencies.
- Color-coding indicates status (e.g., green=succeeded, blue=running, red=failed).
- Clicking a node shows details: schema, comments, run status, data quality metrics.

Data Quality Metrics:

- For each table, DLT displays:
  - Number of input rows.
  - Number of output rows.
  - Number of rows dropped due to failing expectations.
- Detailed breakdown of which expectations passed/failed and how many records were affected.

Run Details Pane:

Overall pipeline status, duration, version.

Links to Spark UI (for advanced debugging).

Error messages if the pipeline failed.

Hands-on Lab: Python Pipeline Creation

MC ID: 7124506 | Parveen KR | Parveen.R@hotmail.com

Lab Objectives

Define Tables

Learn to create Bronze and Silver layer tables using the @dlt.table function in Python.

Ingest Data

Utilize spark.readStream.format("cloudFiles") to effectively load raw data into your pipeline.

Transform Data

Apply various PySpark DataFrame transformations such as withColumn, filter, and select to manipulate data.

Verify Data Quality

Check data in output tables and review data quality metrics to ensure integrity and reliability.

Exam Relevance

Understanding DLT Pipelines

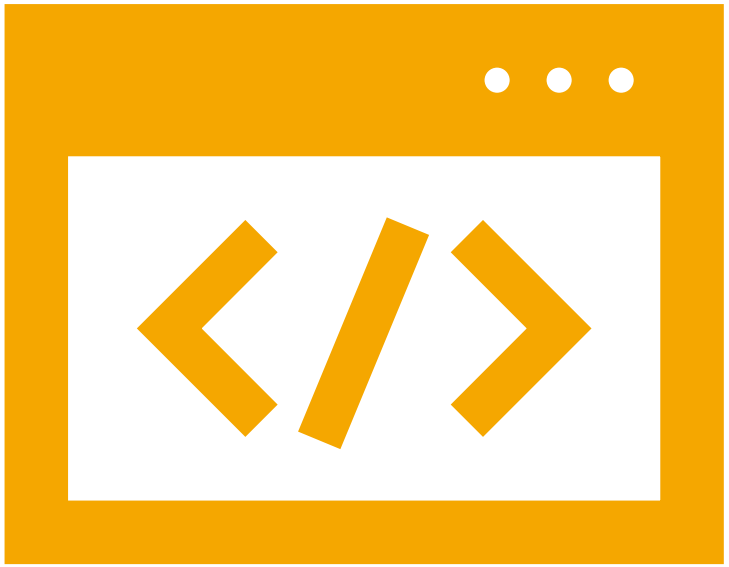
The exam requires a solid understanding of DLT pipelines and their components, which are crucial for data handling.

Python Syntax and Transformations

You need to apply transformations using Python syntax, showcasing your ability to manipulate data effectively.

Data Quality Rules

Implementing data quality rules is essential, ensuring the accuracy and reliability of the data processed through DLT.



Exam Relevance

Essential DLT Import

The import statement for DLT is essential for incorporating necessary features in your Python applications.

Incremental Ingestion Pattern

Using spark.readStream.format("cloudFiles") is a common pattern for ingesting data incrementally.

Schema Inference Location

Specifying cloudFiles.schemaLocation is critical for schema inference and evolution, enhancing data processing.

DataFrame Materialization

Understanding how the function returns the DataFrame to be materialized is vital for data handling.